Start of a great new series!

Fast action – without the dreaded flicker!

BASIC is a simple yet very powerful language found on nearly all home micros, but there are times when it is just not practical to use. One such example is when we try to write fast action games.

Don't get me wrong, Basic is still very good for writing games – as you can see from the quality of those in *The Micro User*. But I'm sure there have been several occasions when you've wanted to move objects smoothly around the screen without the dreaded flicker.

Over the next few months I'll be covering all you need to know about writing machine code games – everthing from sprites to OSBYTEs.

The articles will be written assuming you know the basics of 6502 machine code.

Most games are written in Mode 2 because it offers a medium resolution screen with 16 colours. For this reason nearly all the articles will be devoted to writing games in Mode 2.

However, most of the programs can be adapted to work in other modes as they usually require only a few modifications.

Like all things, we have to start at the bottom and work our way up. This is why the first couple of articles will be concentrating on the configuration of the Mode 2 screen. We need to know how the screen is arranged before we can display objects on it.

I could throw you in the deep end by introducing a full machine code sprite routine, or a program to scroll landscape across the screen, but I'll leave them until we are in a position not only to do so, but also to understand how we're doing it!

As you may already know, the Mode 2 screen uses up 20k of memory to store the display. This 20k is known as the video RAM.

The start of video RAM can be found by printing the value of HIMEM – in Modes 0, 1 and 2 HIMEM is 12288 (&3000 in hex). The last byte of memory used by the video RAM in any mode is &7FFF – just below the sideways ROM area.

The Mode 2 display consists of 32 rows of 20 characters. The video RAM is divided up into strips which go across the screen from left to right, starting at the top left of the screen.

These strips directly correspond to the 32 character rows. Each row or strip of the screen takes up 640 bytes.

Since 20k is equal to 20*1024 bytes (20480), dividing it into 32 rows gives 640 bytes (&280 in hex) per row.

These strips are now split into 80 sets of eight vertical bytes (8*80=640) to give 80 columns. This is getting a bit complex so it's time to simplify things – see Figure I.

This shows the top of the Mode 2 screen – assuming no scrolling has occurred. If it has the start address will be different, but the screen will still be organised in much the same way.

The top left location is &3000 (HIMEM) the next location down is &3001 and so on down the column to location &3007. At this point the next location down is &3280 – note that this is on the next character row.

Surely, it would be more logical if this had been location &3008? Maybe, but the way the screen has been arranged on the BBC Micro is very useful, as we will see in future articles.

If you look back to the first row, to the start of the column right of location &3000 you will see location &3008. Now go down from here and you will find &3009, &300A and so on down to location &300F.

Again, after eight bytes we start a new column on the same row. This is repeated 80 times on each row.

After 80 columns we start at the beginning of the next row down. The second row is arranged in the same way as the first – this time the row starts at location &3280, &3000+(80 columns *

	COLUMN					
	0	1	2	---	78	79
HIMEM→	&3000	&3008	&3010	-	&3270	&3278
	&3001	&3009	&3011	-	&3271	&3279
	&3002	&300A	&3012	-	&3272	&327A
ROW 0	&3003	&300B	&3013	-	&3273	&327B
	&3004	&300C	&3014	-	&3274	&327C
	&3005	&300D	&3015	-	&3275	&327D
	&3006	&300E	&3016	-	&3276	&327E
	&3007	&300F	&3017	-	&3277	&327F
	&3280	&3288	&3290	-	&34F0	&34F8
	&3281	&3289	&3291	-	&34F1	&34F9
	&3282	&328A	&3292	-	&34F2	&34FA
ROW 1	&3283	&328B	&3293	-	&34F3	&34FB
	&3284	&328C	&3294	-	&34F4	&34FC
	&3285	&328D	&3295	-	&34F5	&34FD
	&3286	&328E	&3296	-	&34F6	&34FE
	&3287	&328F	&3297	-	&34F7	&34FF
	&3500	&3508	&3510	-	&3770	&3778
	&3501	&3509	&3511	-	&3771	&3779
	&3502	&350A	&3512	-	&3772	&377A
ROW 2	&3503	&350B	&3513	-	&3773	&377B
	&3504	&350C	&3514	-	&3774	&377C
	&3505	&350D	&3515	-	&3775	&377D
	&3506	&350E	&3516	-	&3776	&377E
	&3507	&350F	&3517	-	&3777	&377F

Figure I

Part One

HOW TO WRITE MACHINE CODE GAMES

By KEVIN EDWARDS

8 bytes)=&3280. That's why the next location down from &3007 is &3280.

It's now time for the first of many programs.

Type in and run Program I. All it is designed to do is store the number 15 in each byte of the video RAM, starting from location &3000 (the top left corner) and ending at &7FFF – if you are prepared to wait that long.

To slow it down a bit I arranged that a key must be pressed before the next location will be changed.

You will notice that a wide yellow pixel appears on the screen each time you press a key – this may be difficult to see on a normal television set. In fact there are two pixels, side by side.

These can be individually addressed with any of the 16 colours available – in this example they are the same colour because the number 15 means display two yellow pixels – we'll see why in a minute. If the number had been different, other colours would appear.

The point is we now know that the colour of the two pixels can be changed by storing different numbers in the video RAM – the number must be in the range 0 to 255 – a byte.

Program II shows what happens when bytes 0 to 255 are stored consecutively in the video RAM – only the first 256 bytes of rows 0,1 and 2 are changed, for speed. You should see three identical coloured sections of rows across the screen.

Program III is identical to Program II, though this time machine code has been used to speed it up.

Here's a brief description of how Program III works:

Line number	
100	Loads the accumulator with the byte that will be stored in the video RAM.
110	Loads the X register with 0. This is used as an offset register.
120	Stores the byte held in the accumulator at location &3000+X, the start of row 0.
130	Stores the byte at location &3280+X, row 1.
140	Stores the byte at location &3500+X, row 2.
150	Increments the offset register by one.
160	Causes a branch if the X register is not 0. If X is zero, the loop is ended and the program returns to Basic because of the RTS in line 170.

Remember that the 6502 registers can only hold a number between 0 and 255. So when the X register contains 255 and an INX is executed, the X register will contain 0. This means that only the first 256 bytes on each line will be changed by the program – 0 to 255 inclusive.

To show how fast machine code is, type in and run Program IV and Run it. This puts the bytes 0 to 255 in all 20k of the video RAM. Press a key to

```
10 REM PROGRAM I
20 MODE 2
30 VDU23;8202;0;0;0;
40 byte=15
50 FOR location=&3000 TO &7FFF
60 ?location=byte
70 REPEAT UNTIL GET
80 NEXT
```

Program I

```
10 REM PROGRAM II
20 MODE 2
30 byte=0
40 REPEAT
50 VDU31,0,6
60 PRINT"Byte = ";byte
70 FOR offset=0 TO &FF
80 offset?&3000=byte
90 offset?&3280=byte
100 offset?&3500=byte
110 NEXT
120 REPEAT UNTIL GET
130 byte=byte+1
140 UNTIL byte>255
```

Program II

```
10 REM PROGRAM III
20 MODE 2
30 byte=0
40 REPEAT
50 VDU31,0,6
60 PRINT"Byte = ";byte
70 FOR pass=0 TO 2 STEP2
80 P%=&C00
90 [OPT pass
100 .Start LDA#byte
110 LDX#0
120 .next STA&3000,X
130 STA&3280,X
140 STA&3500,X
150 INX
160 BNE next
170 RTS
180 ]NEXT pass
190 CALL Start
200 REPEAT UNTIL GET
210 byte=byte+1
220 UNTIL byte>255
```

Program III

```
10 REM PROGRAM IV
20 MODE 2
30 byte=0
40 REPEAT
50 FOR pass=0 TO 2 STEP2
60 P%=&C00
70 [OPT pass
80 .Start LDA#byte
90 LDX#0
100 .next STA&3000,X
110 INX
120 BNE next
130 INC next+2
140 BPL next
150 RTS
160 ]NEXT pass
170 CALL Start
180 VDU31,5,15
190 PRINT"Byte = ";byte
200 REPEAT UNTIL GET
210 byte=byte+1
220 UNTIL byte>255
```

Program IV

From Page 75

increment the value of the byte stored in screen memory.

Line number	
80	In Program IV loads the accumulator with the byte to be stored in the video RAM.
90	Loads the X register with 0 – this is used as an offset.
100	Stores the byte on the screen.
110	Increments the offset register.
120	Branches if the X register is not 0. The branch is not taken until the index register, X, has taken all of the values 0 to 255.
130	Increments the high byte of the screen address being changed.
140	Branches to the next 256 block if the address is still part of the video RAM. The screen ends at location &8000. The high byte of &8000 is negative, because bit 7 is set (&80= %10000000). That's why we only branch if the result is positive.

The RTS in line 150 returns to Basic.

What we must now do is find out the relationship between the byte stored in the video RAM and the colour of the two pixels displayed.

To explain this easily we must use binary numbers.

A nybble, for those who don't know, is a four bit number – sometimes known as as half a byte. A byte consists of two nybbles – the top nybble and the bottom nybble.

Each nybble has 16 different states (0-15, %0000 to %1111). This is very convenient for representing the 16 colours in Mode 2.

The logical way would be to use the top nybble (bits 4-7) of the byte being stored for the left pixel's colour, and the lower bybble (bits 0-3) for the right pixel's colour.

But this isn't the method used. Instead, we have to interweave the bits corresponding to the colours of the two pixels.

This is quite fun, but a pain in the neck when the previously explained method could have been used.

The reason for the interweaving is all to do with the hardware – it would be nice if they considered the people who have to write the software.

If we want to make the left pixel yellow and the right pixel red, we have to calculate the binary number for the colour of the left and right pixel – yellow is %0011 and red is %0001. Table I will help you convert colour numbers into binary.

Now we interweave the bits of the two colours to make up the byte that will be stored in the video RAM. We get four bits from the colour of each pixel to make up eight bits (a byte).

First of all we take the most significant bit (the left-most bit) from the left pixel's colour – 0 in the case of yellow. This will be bit 7 of the byte we are creating.

Now we take the most significant bit from the right pixel's colour – 0 in the case of red. This is bit 6 of the new byte.

We now repeat this procedure for the remaining 6 bits – 3 bits from each colour. Remember to take the first bit from the left colour and the next bit from the right colour.

If this is done correctly you will finish up with the binary number %00001011

0	%0000	Black
1	%0001	Red
2	%0010	Green
3	%0011	Yellow
4	%0100	Blue
5	%0101	Magenta
6	%0110	Cyan
7	%0111	White
8	%1000	Black/White
9	%1001	Red/Cyan
10	%1010	Green/Magenta
11	%1011	Yellow/Blue
12	%1100	Blue/Yellow
13	%1101	Magenta/Green
14	%1110	Cyan/Red
15	%1111	White/Black

Table I: Colour number and binary equivalent

left	right	colour
0	0	Black
2	1	Red
8	4	Green
10	5	Yellow
32	16	Blue
34	17	Magenta
40	20	Cyan
42	21	White
128	64	Black/White
130	65	Red/Cyan
136	68	Green/Magenta
138	69	Yellow/Blue
160	80	Blue/Yellow
162	81	Magenta/Green
168	84	Cyan/Red
170	85	White/Black

Table II: Ready-reference guide for pixel colours

(11 in decimal, &B in hex).

Figure II should help clear the mist – take a look.

If you change the variable *byte* to 11, in Program I, you'll see that the left pixel will be yellow and the right pixel will be red. Hooray! It worked.

To illustrate this further, I've written a program to calculate the byte to be stored given the colour of the left and right pixels.

To ease the confusion, all numbers are displayed in binary. This helps to show how the bits have been interwoven.

Type in Program V and try it. It's all in Basic so I'll leave it up to you to find out how it works.

```
%ABCD -left pixel's colour.
%EFGH -right pixel's colour.

 A B C D
 |E|F|G|H
 ↓↓↓↓↓↓↓↓
%AEBFCGDH = Byte.
```

Figure II: Interweaving pixels

Program VI is the same as Program V but colour blocks now replace the binary numbers.

Table II is a ready-reference guide to pixel colour combinations. For example, to calculate the byte required to produce a yellow pixel on the left and a blue pixel on the right you add up the numbers 10 and 16, resulting in 26. It's as easy as that.

So far we've concentrated on how the screen is mapped, and how each pixel can be set in any colour. This is vital knowledge whatever you're doing.

You see, graphics are all to do with making shapes on the screen by changing the contents of the video RAM – it's just the same if you want to display a space invader or the letter Z.

Most of this month's programs are in Basic. This may seem a bit weird when the article is all about machine code games but it is easier to explain things initially in Basic.

Don't worry, next month's article jumps deeper into the world of machine code, so swot up your 6502 as soon as possible.

See you next month.

```
   10 REM PROGRAM V
   20 REM By Kevin Edwards
   30 MODE 7
   40 no=FALSE:yes=TRUE
   50 want_space=no
   60 REPEAT
   70 INPUT'''"Enter colour for left
pixel (0-15)",left
   80 UNTIL left>=0 AND left<16
   90 PRINT''left;" = %";
  100 PROCbinary(left)
  110 left_nibble$=nibble$
  120 REPEAT
  130 INPUT'''"Enter colour for right
 pixel (0-15)",right
  140 UNTIL right>=0 AND right<16
  150 PRINT''right;" = %";
  160 PROCbinary(right)
  170 right_nibble$=nibble$
  180 PRINT''TAB(10);
  190 want_space=yes
  200 PROCbinary(left)
  210 PRINT'TAB(11);
  220 PROCbinary(right)
  230 PRINT'TAB(11);"--------"
  240 PRINTTAB(10);"%";
  250 byte$=""
  260 FOR bit=1 TO 4
  270 byte$=byte$+CHR$(ASC(MID$(left_
nibble$,bit,1)))
  280 byte$=byte$+CHR$(ASC(MID$(right
_nibble$,bit,1)))
  290 NEXT
  300 PRINT;byte$
  310 PRINTTAB(11);"--------"
  320 byte=0
  330 FOR bit=8 TO 1 STEP -1
  340 IF ASC(MID$(byte$,bit,1))=ASC"1
" THEN byte=byte+(2^(8-bit))
  350 NEXT
  360 PRINT''TAB(7);"Byte = ";byte;"
or &";~byte
  370 END
  380 DEFPROCbinary(nibble)
  390 nibble$=""
  400 FOR bit=3 TO 0 STEP -1
  410 IF want_space VDU ASC" "
  420 IF nibble AND (2^bit) VDU ASC"1
":bit$="1" ELSE VDU ASC"0":bit$="0"
  430 nibble$=nibble$+bit$
  440 NEXT
  450 ENDPROC
```

Program V

```
   10 REM PROGRAM VI
   20 REM By Kevin Edwards
   30 @%=8
   40 VDU23,224,255,255,255,255,255,2
55,255,255
   50 DIM pattern(1,15)
   60 FOR left_right=0 TO 1
   70 FOR colour=0 TO 15
   80 READ pattern(left_right,colour)
   90 NEXT
  100 NEXT
  110 MODE 2
  120 FOR loop=0 TO 15
  130 COLOUR loop
  140 VDU 224,32
  150 COLOUR 7
  160 PRINT;loop,;
  170 NEXT
  180 INPUT''"Left  ...",Left
  190 IF Left<0 OR Left>15 VDU7:GOTO
180
  200 INPUT''"Right ...",Right
  210 IF Right<0 OR Right>15 VDU7:GOT
O 200
  220 PRINT''
  230 COLOUR Left
  240 VDU 224
  250 COLOUR Right
  260 VDU 224
  270 COLOUR 7
  280 PRINT;" = ";
  290 Byte=pattern(0,Left)+pattern(1,
Right)
  300 PRINT;Byte;" or &";~Byte
  310 *FX15
  320 PRINT''"Press SPACE to cont"
  330 REPEAT UNTIL GET=32
  340 GOTO 110
  350 REM Data for left-hand pixel
  360 DATA0,2,8,10,32,34,40,42,128,13
0,136,138,160,162,168,170
  370 REM Data for right-hand pixel
  380 DATA0,1,4,5,16,17,20,21,64,65,6
8,69,80,81,84,85
```

Program VI

Follow in the footsteps of Elite with the EOR method

WE looked at the configuration of the Mode 2 screen last month and found out the colour of any pixel could be changed by storing different bytes in the relevant video RAM locations.

Now we'll go a bit further by joining several pixels together to form a character.

We know that the Mode 2 screen consists of 32 strips or character rows. These are split into 80 vertical rows one byte in width, each being 8 pixels high and 2 pixels wide (2 pixels per byte in Mode 2).

From this we can see that the number of characters per row in Mode 2 is 20. The 160 pixels horizontally (80 columns * 2 pixels), divided by 8 (width of a character in pixels) gives 20, therefore the number of bytes required to store a character is 32, not 64 as you might expect (8*8).

This is because the character is 8 bytes by 4 – remember that each horizontal byte corresponds to two pixels.

Have a look at Diagram I. This shows the size of each character cell.

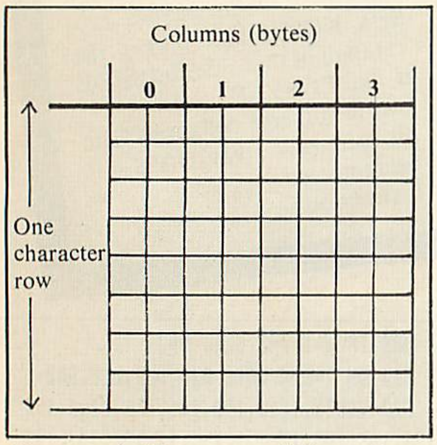


Diagram I: Character cell dimensions

You may think 24 bytes for one character is a bit wasteful when the VDU 23 command only requires eight byte parameters to define a shape of the same size. What you must remember is that a character defined by VDU 23 cannot be multi-coloured.

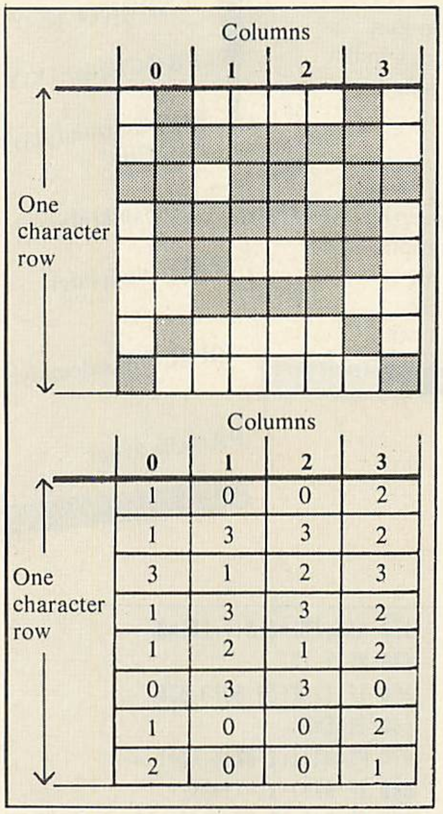
Let's design a space invader, 8 pixels by 8. For this we will have to calculate the 24 bytes that represent the shape and colour we want the invader. Diagram II shows the bytes required to produce a red space invader.

Table I in last month's article will help you understand how the bytes were calculated.

What we need to do now is find a way of displaying the space invader. Program I does this. The bytes of the video RAM are arranged in 8 byte columns. As we work our way down each column the memory location increases. So it would seem logical to store the data for the character in the same way.

Storing the data like this allows us to use a simple indexing method to copy the bytes from outside the video RAM into the video RAM. The description of Program I gives the full details.

Diagram III shows the data for a yellow face. Again, the face is 8 pixels by 8 – a character. For simplicity I've designed characters with only one foreground colour.



0	1	2	3
1	0	0	2
1	3	3	2
3	1	2	3
1	3	3	2
1	2	1	2
0	3	3	0
1	0	0	2
2	0	0	1

Diagram II: Parameters for a red invader

You can use as many colours as you want for your own characters. The best way to design them is to use squared paper and felt tip pens. Then once you've finished designing your character you can calculate the bytes required to display the new shape. Substituting these with the data in Program I will allow you to see the results.

A future article will include a multi-coloured character definer. This will help speed up the definition of characters.

Program I can also be used to display

```
 10 REM PROGRAM I
 20 MODE 2
 30 FOR pass=0 TO 2 STEP 2
 40 P%=&B00
 50 [OPT pass
 60 .start LDY#31
 70 .loop1 LDA&C00,Y
 80 .display STA&5400,Y
 90 DEY
100 BPL loop1
110 RTS
120 ]NEXT
130 FOR loop=0 TO 31
140 READ data
150 loop?&C00=data
160 NEXT
170 CALL start
180 END
190 REM Space Invader data
200 DATA1,1,3,1,1,0,1,2
210 DATA0,3,1,3,2,3,0,0
220 DATA0,3,2,3,1,3,0,0
230 DATA2,2,3,2,2,0,2,1
```

Program I: Plotting characters

Part two

HOW TO WRITE MACHINE CODE GAMES

By KEVIN EDWARDS

the face. In fact, any data for an 8 by 8 character will work with Program I. All you need to do is change the DATA lines to those shown in Listing I. It's so simple – and so quick!

```
190 REM Face data
200 DATA5,15,10,15,15,15,15,5
210 DATA15,15,5,15,15,5,0,15
220 DATA15,15,10,15,15,10,0,15
230 DATA10,15,5,15,15,15,15,10
```

Listing I

Program I's space invader is stored in the video RAM starting at location &5400 and ending at location &541F – 32 bytes. If we wanted to put another invader on the screen to the right of the existing one we would store the shape staring from location &5420 (&5400 + 32=&5420).

The 32 corresponds to the number of bytes needed to accommodate one character. We add this to the old start location to move the start address to the next character cell – the screen locations increase as you move to the right and decrease as you go to the left. Therefore to move one character cell to the left we would subtract 32 from the original start location.

Program II is similar to Program I except this time the start location is increased by 32 after each character is displayed and we also start at the top left of the screen. This moves the shape right by one character cell but leaves the previous shape on the screen.

All we need to do now is delete the previous character to produce a moving object – for the majority of cases you will not want to leave a trail behind a moving character.

The easiest way to delete a shape is to store zeros in the video RAM which contains the character data.

Storing zero produces two black pixels on the screen. Since black is the normal background colour, the previous shape disappears – printing a space on the screen has exactly the same effect.

The problem with deleting a shape in this way is that the background data is lost. A good example where problems would occur is in the arcade game Pacman.

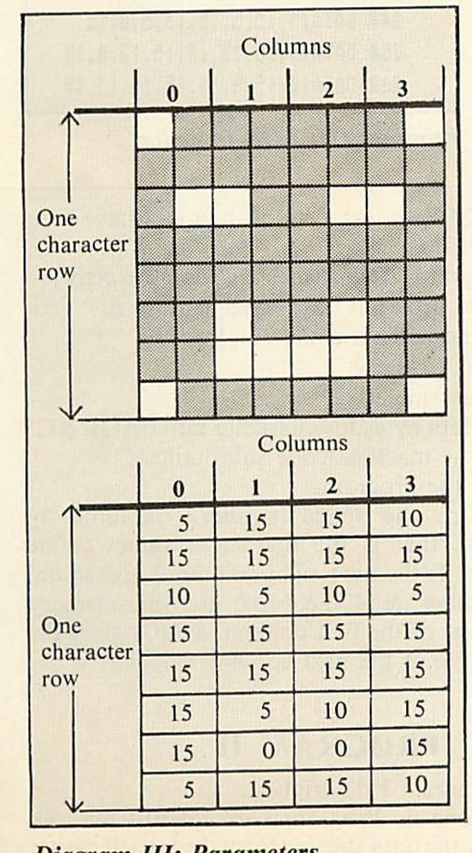
Here monsters move around a maze which is filled with dots. Your aim is to guide another character around the maze so that he collects all the dots by moving over them. If the monster characters were deleted by storing zeros in the video RAM the dots would be destroyed as soon as one of the monsters passed over them. That's rather pointless when the idea of the game is to eat up all the dots on the screen, for the monsters do it for you.

The problem is solved by using the logical operator EOR (Exclusive OR). This is very useful because its effect on numbers is reversible.

For example, consider two numbers X and Y. We shall define Z as being X EOR Y. The effect of EOR also allows us to say Y EOR Z=X and X EOR Z=Y. Using numbers, 1 EOR 2=3, 2 EOR 3=1 and 1 EOR 3=2.

The EOR function works by comparing corresponding bits of two bytes with each other, and depending on the state of the two bits produces a binary digit. This bit is used to build up a new byte – the resultant byte.

Table I shows the resultant bit for the



0	1	2	3
5	15	15	10
15	15	15	15
10	5	10	5
15	15	15	15
15	15	15	15
15	5	10	15
15	0	0	15
5	15	15	10

Diagram III: Parameters for a yellow face

```
10 REM PROGRAM II
20 MODE 2
30 FOR pass=0 TO 2 STEP 2
40 P%=&B00
50 [OPT pass
60 .start LDY#31
70 .loop1 LDA&C00,Y
80 .display STA&3000,Y
90 DEY
100 BPL loop1
110 LDA display+1
120 CLC
130 ADC#32
140 STA display+1
150 BCC no_carry
160 INC display+2
170 .no_carry RTS
180 ]NEXT
190 FOR loop=0 TO 31
200 READ data
210 loop?&C00=data
220 NEXT
230 FOR loop=1 TO 200
240 CALL start
250 REPEAT UNTIL GET
260 NEXT
270 END
280 REM Space Invader data
290 DATA1,1,3,1,1,0,1,2
300 DATA0,3,1,3,2,3,0,0
310 DATA0,3,2,3,1,3,0,0
320 DATA2,2,3,2,2,0,2,1
```

Program II: 'Moving' shapes

Bit n from 1st byte	Bit n from 2nd byte	Result bit
0	0	0
0	1	1
1	0	1
1	1	0

Table I: Truth table for EOR

From Page 75

four possible conditions of the two bits being compared. In the table, n is the nth bit of the two bytes being compared.

To show the EOR command in action we will EOR the numbers &9F and &54.

The first thing we do is convert them into binary. Now we take the first bit from each byte and find the resultant bit – using the "truth table". This bit is the first bit of the resultant byte. We now do exactly the same for the remaining seven bits to complete the new byte.

If this seems rather complex you will be relieved to know that the 6502 has an EOR command that does all the work for you. Diagram IV shows how to calculate &9F EOR &54 by hand.

To explain how EOR is used, consider &9F as being the background byte and &54 as the character byte. When the character passes over the background, and we use EOR to calculate the resultant byte which will be stored in the video RAM, we get &CB – &54 EOR &9F.

When we want to move the character to a new position we EOR the screen byte (&CB) with the character byte (&54) to move it off the screen.

```
&9F = % 10011111
&54 = % 01010100
        --------
      % 11001011 = &CB or 195
        --------
```

Diagram IV: EORing two bytes

'You will be relieved to know that the 6502 has an EOR command that does all the work for you...'

If all goes well we get the original background byte (&9F). This means that we've moved a character over a background without disturbing it. Problem solved? Well almost.

So far we've dealt with EOR for one byte. So for a character which is several bytes in size we must perform the EOR on each byte we store in the video RAM.

EOR has two drawbacks. When an object passes over another the resultant byte, and therefore colour, is not the same as the background byte or the character byte – this means that a colour change occurs.

In the previous example we get &CB, which is totally different to the original bytes used.

The other problem with EOR is that when the character byte and the background byte are the same the

```
 10 REM PROGRAM III
 20 MODE 2
 30 FOR pass=0 TO 2 STEP 2
 40 P%=&B00
 50 [OPT pass
 60 .start LDY#31
 70 .loop1 LDA&C00,Y
 80 EOR&5400,Y
 90 .display STA&5400,Y
100 DEY
110 BPL loop1
120 RTS
130 ]NEXT
140 FOR loop=0 TO 31
150 READ data
160 loop?&C00=data
170 NEXT
180 REPEAT
190 CALL start
200 REPEAT UNTIL GET
210 UNTIL 1=2
220 REM Face data
230 DATA5,15,10,15,15,15,15,5
240 DATA15,15,5,15,15,5,0,15
250 DATA15,15,10,15,15,10,0,15
260 DATA10,15,5,15,15,15,15,10
```

Program III: EOR in action

PROGRAM I

30-50	Select options for assembler.
60	Loads the Y register with 31. This is used as an offset which decrements from the last byte of the character's position to the first byte of the character's position.
70	Reads a data byte from location &C00+Y. The last data byte is read first and the first data byte is read last.
80	Stores the data byte in the video RAM at location &5400+Y. Again working from the end to the start location.
90	Decrements the offset register, Y.
100	Branches to do the next data byte if the Y register is still positive – in the range 0-127. This branch is not taken when the Y register becomes &FF (255 in decimal) – this means that the 32 data bytes have been stored in the video RAM, and Y has taken the values 31 to 0. It is easier to use an offset register which decrements from the end to zero than one which increments from 0 to the end because you do not need to use a CMP command or, in this case, CPY to test if all of the bytes have been copied. Program IA illustrates this, doing exactly the same as program I but starting at the first data byte and ending at the 32nd one.
110	Returns to Basic.
120	Ends assembler.
130-160	Store data bytes for character into PAGE &C.
170	Calls the machine code subroutine.
180	ENDs the program.
190-230	Data for the space invader – column by column, that is the first eight values define column 1, the next eight column 2 and so on. Remember &5400-&5407 are the memory locations of the first column. &5408 gives the start byte of the next column.

PROGRAM II

10-50	See program I description.
60-100	The same as Program I except this time we display the data starting from the top left corner (&3000).

resultant byte will always be zero – the background colour, black.

One byte which matches the background will not be very noticeable. But when two identical characters are directly on top of one another the result will be two invisible characters. Rather hard to see.

Despite these disadvantages, many commercial games use this method to display characters on the screen.

To show EOR in action try Program III. All this does is use the EOR method to display a face character on the screen.

Initially the face appears in the centre of the screen. Since the background bytes are all zero and EORing any number with 0 leaves the number unchanged the data stored in the video RAM will be identical to the character data.

Pressing a key causes the same shape to be placed on the screen again. Since EOR has been used to calculate the resultant bytes, the face disappears. This is because EORing a byte with itself results in zero – this was the original background colour.

So from this we can see that calling the routine the first time displays the face and calling the same routine again erases the face. In other words, the character display routine is also a character delete routine – two for the price on one!

It is for this reason that program writers use the EOR method to display shapes – it only needs one routine. Even Elite uses EOR when it plots lines on the screen.

Try adding the following line to program III:

```
175 VDU 31,8,14,ASC"A"
```

This displays the letter A underneath the face character. Any points of the letter A and the face which overlap produce a new colour, blue because of the effect EOR has. These points will be blue (colour 4), because, 3 (yellow face) EOR 7 (white letter A) = 4.

Pressing Space deletes the face and leaves the letter A on the screen – this time the A will be in its original colour, white.

● *That's enough for this month. Next month we are going to move shapes around the screen.*

```
10 REM PROGRAM IA
20 MODE 2
30 FOR pass=0 TO 2 STEP 2
40 P%=&B00
50 [OPT pass
60 .start LDY#0
70 .loop1 LDA&C00,Y
80 .display STA&5400,Y
90 INY
100 CPY#32
110 BNE loop1
120 RTS
130 ]NEXT
140 FOR loop=0 TO 31
150 READ data
160 loop?&C00=data
170 NEXT
180 CALL start
190 END
200 REM Space Invader data
210 DATA1,1,3,1,1,0,1,2
220 DATA0,3,1,3,2,3,0,0
230 DATA0,3,2,3,1,3,0,0
240 DATA2,2,3,2,2,0,2,1
```

Program IA: An alternative way of plotting a character

110-140	Add 32 to the screen location to move the start location to the next character cell. This method involves self-altering code. By this I mean that bytes are "poked" into the machine code routine by the machine code routine. In this example it adds 32 to the low byte of the screen address.
150	Branches if the addition doesn't produce a carry.
160	Increments the high byte of the screen location. Again, this alters the machine code program by adding one to the screen's high byte.
170	Returns to Basic.
180	ENDs assembler.
190-220	Read and store the graphic data.
230-260	Loops round until 100 space invaders have been displayed.
240	Calls the routine which displays the space invader.
250	Waits until a key is pressed.
260	End of FOR TO loop.
270	End of program.
280-320	Space invader graphic data.

PROGRAM III

30-50	Set up assembler.
60	Sets offset register, Y, to 31.
70	Reads data byte into the accumulator from location &C00+Y.
80	EORs the accumulator with the contents of the destination location of the byte – the result is left in the accumulator.
90	Stores the accumulator in the video RAM at location &5400+Y.
100	Decrements the offset register, Y.
110	Branches if the character is not complete.
120	Returns to Basic.
130	Ends assembler.
140-170	Read and store the face data.
180	Start of infinite loop.
190	Calls the machine code routine to display the face or delete the face. Remember that the same routine can display a character or delete a character – it's a property of EOR.
200	Waits until a key is pressed.
210	Repeats the loop, since 1 will never equal 2.
220-260	Face graphic data.

Part three

HOW TO WRITE MACHINE CODE GAMES

By KEVIN EDWARDS

Face up to it, you don't need to move one pixel at a time

LAST month we found that moving objects around the Mode 2 screen – left and right – was quite easy. The main problem was how to delete the character. We solved it by using the EOR function, which provided us with a method of displaying and deleting a character by using just one routine.

This month we're going to move shapes up, down, left and right. In the process we'll also find out how difficult it is to move a shape up and down the screen one pixel at a time. So far, to display characters we've used an indexing method when accessing the screen.

The base location is used to indicate the start position of the character – its top left corner. Changing this base location allows us to move the object to other parts of the screen.

As we saw last time, adding 32 to the base location moves an object right by one character cell, and subtracting 32 from the base location moves the shape left one character cell.

This may satisfy certain needs, but for most cases finer movement is needed.

To do this we subtract or add eight to the base location. Subtracting eight moves the character left by two pixels and adding eight moves the character right by two pixels.

Let me stress that adding or subtracting eight moves the base location to an adjacent column. Remember, each character row has 80 columns of eight vertical bytes – each column byte corresponds to two pixels. That's why the object moves two pixels when eight is added or subtracted.

Moving an object left or right by one pixel is very awkward. This is because each memory location represents two pixels.

It means that in addition to changing the base location, the character data must also be changed – shifted in the appropriate direction by one pixel.

The easiest way to shift the data is to store the character data again – displaced by one pixel.

The result is that twice as much memory is needed to store the character data. In Mode 2, memory is very valuable and so it's always best to avoid repeating data.

Software techniques can be used to shift characters across by one pixel, but these are very time consuming and so are rarely used – speed is considered more important than looks.

Nearly all commercial games written in Mode 2 move characters left and right by two pixels, so don't get paranoid because you haven't got true pixel movement. It's just not worth the extra memory and effort required, and in any case, moving two pixels at a time produces very smooth movement.

If you don't believe me try Program I. All you'll see is a face character go across the screen – left to right.

Program II allows you to move the face character left or right. The direction is controlled by pressing the Z and X keys – Z to move left and X to move right.

We can now move a character left or right. The next thing to do is move it up or down. I'm afraid this is the most complex part of character movement.

It's simple to move a character up or down by one character line. To do this, 640 must be subtracted or added to the base location. Adding 640 moves down one character line and subtracting 640 moves up one line. Why 640?

Each character line has 80 columns of eight bytes. So the next line begins at the previous line's base location plus 640 (80 * 8).

The start locations of the first three

→

```
10 REM PROGRAM II
20 MODE 2
30 FOR pass=0 TO 2 STEP 2
40 P%=&B00
50 [OPT pass
60 .left LDA#19:JSR&FFF4
70 JSR character
80 LDA display+1
90 SEC
100 SBC#8
110 STA display+1
120 STA eor_screen+1
130 BCS no_decrement
140 DEC display+2
150 DEC eor_screen+2
160 .no_decrement JSR character
170 RTS
180 .right LDA#19:JSR&FFF4
190 JSR character
200 LDA display+1
210 CLC
220 ADC#8
230 STA display+1
240 STA eor_screen+1
250 BCC no_increment
260 INC display+2
270 INC eor_screen+2
```

Program II

```
10 REM PROGRAM I
20 MODE 2
30 FOR pass=0 TO 2 STEP 2
40 P%=&B00
50 [OPT pass
60 .right LDA#19:JSR&FFF4
70 JSR character
80 LDA display+1
90 CLC
100 ADC#8
110 STA display+1
120 STA eor_screen+1
130 BCC no_increment
140 INC display+2
150 INC eor_screen+2
160 .no_increment JSR character
170 RTS
180 .character LDY#31
190 .getbyte LDA&C00,Y
200 .eor_screen EOR&5300,Y
210 .display STA&5300,Y
220 DEY
230 BPL getbyte
240 RTS
250 ]NEXT
260 FOR loop=0 TO 31
270 READ data
280 loop?&C00=data
290 NEXT
300 CALL character
310 FOR loop=1 TO 75
320 CALL right
330 NEXT
340 END
350 REM Face data
360 DATA5,15,10,15,15,15,15,5
370 DATA15,15,5,15,15,5,0,15
380 DATA15,15,10,15,15,10,0,15
390 DATA10,15,5,15,15,15,15,10
```

Line No.	
10-50	Enter assembler.
60	Main entry point. This line executes a *FX 19 command (wait for vertical sync.). Used as a time delay to produce smooth movement without flicker.
70	Delete the face character. Remember, calling an EOR character routine the second time deletes the character.
80-150	Add eight to the base address – move the character right by two pixels.
160	Display the face in its new screen position.
170	End of the main routine.
180-240	Subroutine to display the face on the screen.
250	Exit assembler.
260-290	Read and store the face data in page &C.
300	Put face on the screen – initial set-up.
310-330	Move the face right through 75 positions.
340	End of program.
350-390	Face data.

Description of Program I

```
280 .no_increment JSR character
290 RTS
300 .character LDY#31
310 .getbyte LDA&C00,Y
320 .eor_screen EOR&5300,Y
330 .display STA&5300,Y
340 DEY
350 BPL getbyte
360 RTS
370 ]NEXT
380 FOR loop=0 TO 31
390 READ data
400 loop?&C00=data
410 NEXT
420 X=0
430 CALL character
440 REPEAT
450 IF INKEY(-98) AND X>0 CALL left
:X=X-1
460 IF INKEY(-67) AND X<76 CALL rig
ht:X=X+1
470 UNTIL 1=2
480 REM Face data
490 DATA5,15,10,15,15,15,15,5
500 DATA15,15,5,15,15,5,0,15
510 DATA15,15,10,15,15,10,0,15
520 DATA10,15,5,15,15,15,15,10
```

Line No.	
10-50	Enter assembler.
60	Wait for vertical sync.
70	Delete the face.
80-150	Subtract eight from the base location – move the character left by two pixels.
160	Display the character in it's new position.
170	End of routine.
180	Wait for vertical sync.
190	Delete the face.
200-270	Add eight to the base location – move it right by two pixels.
280	Display the face.
290	End of routine.
300-360	Subroutine to display the face on the screen.
370	Exit assembler.
380-410	Read and store graphic data.
420	X coordinate=0.
430	Display the character.
440	Start of an infinite loop.
450	Check if Z is pressed. Move the character left as long as the X coordinate remains on the same character row. And then decrease the X coordinate.
460	Check if X is pressed. Move the character right as long as the X coordinate remains on the same character row. And then increase the X coordinate.
470	Loop round.
480-520	Face data.

Description of Program II

```
  10 REM PROGRAM III
  20 MODE 2
  30 FOR pass=0 TO 2 STEP 2
  40 P%=&B00
  50 [OPT pass
  60 .up LDA#19:JSR&FFF4
  70 JSR character
  80 LDA display+1
  90 SEC
  100 SBC#&80
  110 STA display+1
  120 STA eor_screen+1
  130 LDA display+2
  140 SBC#2
  150 STA display+2
  160 STA eor_screen+2
  170 JSR character
  180 RTS
  190 .down LDA#19:JSR&FFF4
  200 JSR character
  210 LDA display+1
  220 CLC
  230 ADC#&80
  240 STA display+1
  250 STA eor_screen+1
  260 LDA display+2
  270 ADC#2
  280 STA display+2
  290 STA eor_screen+2
  300 JSR character
  310 RTS
  320 .character LDY#31
  330 .getbyte LDA&C00,Y
  340 .eor_screen EOR&3000,Y
  350 .display STA&3000,Y
  360 DEY
  370 BPL getbyte
  380 RTS
  390 ]NEXT
  400 FOR loop=0 TO 31
  410 READ data
  420 loop?&C00=data
  430 NEXT
  440 Y=0
  450 CALL character
  460 REPEAT
  470 IF INKEY(-73) AND Y>0 CALL up:Y
=Y-1
  480 IF INKEY(-105) AND Y<31 CALL do
wn:Y=Y+1
  490 UNTIL 1=2
  500 REM Face data
  510 DATA5,15,10,15,15,15,15,5
  520 DATA15,15,5,15,15,5,0,15
  530 DATA15,15,10,15,15,10,0,15
  540 DATA10,15,5,15,15,15,15,10
```

Program III

Line No.	
10-50	Enter assembler.
60	Wait for vertical sync.
70	Delete character.
80-160	Subtract &280 from the base location – move the character up one line.
170	Display the character.
180	Exit.
190	Wait for vertical sync.
200	Delete the character.
210-290	Add &280 to the base location – move the character down one line.
300	Display the character.
310	Exit.
320-380	Subroutine to display the character on the screen.
390	Exit assembler.
400-430	Read and store the face data.
440	Y coordinate=0.
450	Display the face.
460	Start of loop.
470	Check the : key. Move the character up if the key is pressed and the character remains on the screen. And then increment the Y coordinate.
480	Check the / key. Move the character down if the key is pressed and the character remains on the screen. And then decrement the Y coordinate.

Description of Program III

From Page 104

character rows are &3000, &3280 and &3500 respectively. Each increases by &280 as you go down the screen. The clever among you will know that &280=640.

Also notice that the difference in memory locations between the same position on ANY adjacent character lines is always &280.

Program III allows you to move a character up and down the screen, one line at a time. Use / to move the character down and : to move it up.

Subtracting or adding 640 from the base location produces character movement up or down. What do we do if we want to move the character up or down by one pixel? There is no simple solution.

To simplify things we'll forget moving a whole character and move a single byte instead. First we'll move a byte down the screen.

You'll find it a lot easier to understand the following information if you read it in conjunction with Diagram I.

Initially, the screen location we store the byte in is &3000 – the top left of the screen. To move the byte down by one pixel we increment the screen location by one – to &3001. All goes fine, incrementing the location until we reach location &3007 – the bottom of the character row.

At this point incrementing the screen location to &3008 moves it to the top of the next column on the same line. In fact, the location we want to store the

	COLUMN					
	0	1	2	...	78	79
HIMEM	&3000	&3008	&3010	...	&3270	&3278
	&3001	&3009	&3011	...	&3271	&3279
	&3002	&300A	&3012	...	&3272	&327A
ROW 0	&3003	&300B	&3013	...	&3273	&327B
	&3004	&300C	&3014	...	&3274	&327C
	&3005	&300D	&3015	...	&3275	&327D
	&3006	&300E	&3016	...	&3276	&327E
	&3007	&300F	&3017	...	&3277	&327F
	&3280	&3288	&3290	...	&34F0	&34F8
	&3281	&3289	&3291	...	&34F1	&34F9
	&3282	&328A	&3292	...	&34F2	&34FA
ROW 1	&3283	&328B	&3293	...	&34F3	&34FB
	&3284	&328C	&3294	...	&34F4	&34FC
	&3285	&328D	&3295	...	&34F5	&34FD
	&3286	&328E	&3296	...	&34F6	&34FE
	&3287	&328F	&3297	...	&34F7	&34FF
	&3500	&3508	&3510	...	&3770	&3778
	&3501	&3509	&3511	...	&3771	&3779
	&3502	&350A	&3512	...	&3772	&377A
ROW 2	&3503	&350B	&3513	...	&3773	&377B
	&3504	&350C	&3514	...	&3774	&377C
	&3505	&350D	&3515	...	&3775	&377D
	&3506	&350E	&3516	...	&3776	&377E
	&3507	&350F	&3517	...	&3777	&377F

Diagram I

```
10 REM PROGRAM IV
20 REM By Kevin Edwards
30 MODE 2
40 FOR pass=0 TO 2 STEP 2
50 P%=&B00
60 [OPT pass
70 .start LDA#63
80 .display STA&3000
90 LDA display+1
100 AND#7
110 CMP#7
120 BEQ bottom_of_column
130 INC display+1
140 RTS
150 .bottom_of_column LDA display+1
160 CLC
170 ADC#&79
180 STA display+1
190 LDA display+2
200 ADC#2
210 STA display+2
220 RTS
230 ]NEXT
240 FOR loop=1 TO 100
250 CALL start
260 REPEAT UNTIL GET
270 NEXT
```

byte in – to move it down one pixel – is &3280 and not &3008.

We now have a problem. How can we tell if we increment the screen location by one or add a much larger number to it?

The clue is that we increment the byte by one only if the current screen location is not at the bottom of the character row and we add &279 if it is at the bottom of the character row – this moves the byte to the top of the next character row.

But how do we know if a screen location is at the bottom of the character row? The answer can be found by using the logical operator AND.

Figure I is a list of some of the locations at the bottom of character row 0. Each location is followed by its binary equivalent.

Have you noticed a pattern? If not, take a look at the binary patterns for the low bytes of the screen locations.

The answer is that the three least significant bits of the low byte of the screen address are all set to 1.

So if bit 0=1 AND bit 1=1 AND bit 2=1 the screen location is at the bottom of the character row.

In practice, we AND the screen location with 7 – we get seven from 4+2+1, bit 2,1 and 0 being set. If the resultant byte is seven then the first three bits of the low byte must be set. In which case the location is at the bottom

```
Screen      high       low
location    byte       byte

&3007 = % 00110000  00000111
&300F = % 00110000  00001111
&3017 = % 00110000  00010111
&301F = % 00110000  00011111
&3027 = % 00110000  00100111
   .          .          .
   .          .          .
&3277 = % 00110010  01110111
&327F = % 00110010  01111111
```

Figure I

Line No.	
10-60	Enter assembler.
70	Load the accumulator with the byte to be stored in the video RAM.
80	Store the accumulator in the video RAM – display the byte (two white pixels).
90-110	Check if the location of the byte displayed is at the bottom of a character row.
120	Branch if it is at the bottom of a character row.
130	Increment the screen location – move the byte down one pixel.
140	Exit.
150-210	Add &279 to the screen location – this moves the byte to the top of the next character row.
220	Exit.
230	Exit assembler.
240	Start of loop.
250	CALL the routine.
260	Wait until a key is pressed.
270	End of loop.

Description of Program IV

of a character row. In assembly, the required check would look something like this:

```
.down_pixel      LDA screenlow
                 AND #7
                 CMP #7
                 BNE not_at_bottom
                 LDA screenlow
                 CLC
                 ADC #&79
                 STA screenlow
                 LDA screenhigh
                 ADC #2
                 STA screenhigh
                 JMP rest_of_program
.not_at_bottom   INC screenlow
.rest_of_program
```

Program IV moves a byte down the screen one pixel at a time. Press a key to continue the movement. Moving shapes up the screen by a pixel is very similar to moving it down.

A byte can be moved up by decrementing the screen address. When the byte reaches the top of a character row a much larger number must be subtracted from the screen location – this larger number is &279.

Now we have to find a way of testing whether a screen location is at the top of a character row. Again, we do this by using the AND function. Consider the following locations which are all at the top of character row 1:

```
Screen      high       low
location    byte       byte

&3280 = % 00110010  10000000
&3288 = % 00110010  10001000
&3290 = % 00110010  10010000
&3298 = % 00110010  10011000
&32A0 = % 00110010  10100000
   .          .          .
   .          .          .
&34F0 = % 00110100  11110000
&34F8 = % 00110100  11111000
          ------------------
          ????????  ?????000
          ------------------
```

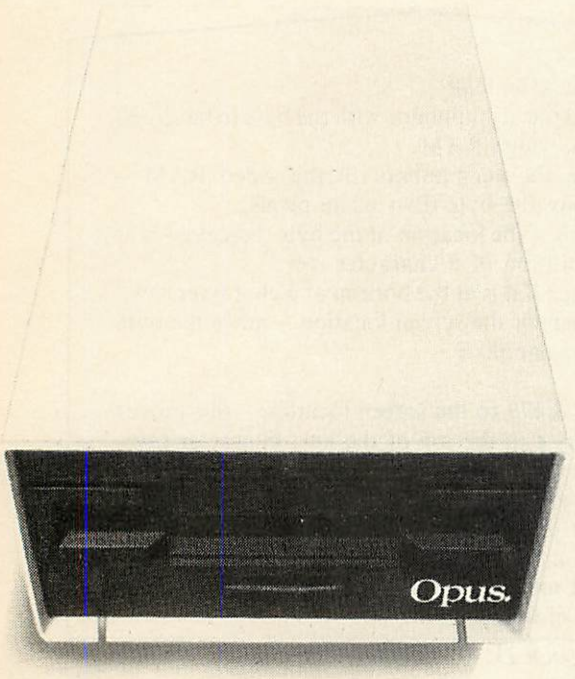
Again bits 0,1 and 2 of the screen address low byte give us the answer – they are all zero this time.

So if bit 0=0 AND bit 1=0 AND bit 2=0 the screen location is at the top of a character row.

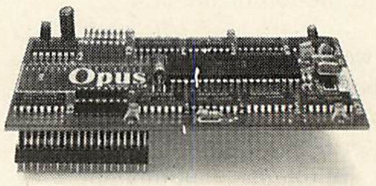
In machine code we AND the low byte with 7, and if the result is 0 then the screen location is at the top of the character row, otherwise the location is

→

WE'RE ASKING FOR LESS. AND GIVING YOU MORE.



THE OPUS 3" DISC DRIVE
NOW ONLY
£199.95 (INC. VAT.)



WITH A DOUBLE DENSITY DISC
INTERFACE (WORTH £100)
FREE.

For only £199.95 you can have one of the best bargains available.

An Opus 500K double sided disc drive and an Acorn compatible double density disc interface. That's over £100 off the normal retail price.

But the bargain doesn't stop there. Because these drives have the Opus name for reliability. Which is why we have the confidence to give a <u>two</u> year guarantee on all our drives.

The reason we can be so confident is that our drives are tested to the limit – running for over 8,000 hours. That's a year of constant use without failure.

To make sure you're not disappointed, complete the coupon and send it off today.

**GENEROUS EDUCATION DISCOUNTS GIVEN.
PLEASE PHONE FOR DETAILS.**

OPUS SUPPLIES LTD.
158 Camberwell Road, London SE5 0EE.
Opening hours:
9.00-5.30pm Monday to Friday.

01-701 8668
01-703 6155
Telex: 295931

GUARANTEED 48 HOUR COURIER DELIVERY. NATIONWIDE.

To: Opus Supplies Ltd., 158 Camberwell Road, London SE5 0EE. Please rush me the following:
(ALL PRICES INCLUDE VAT & CARRIAGE.)

Quantity	Description	Price
	TOTAL	

I enclose a cheque for £ ____
Or please debit my credit card account with the amount of
£ ____
My Access/Barclaycard (please tick) No. is ____

Name ____
Address ____

Telephone ____

Opus.
Opus Supplies Ltd.

BU16

PLEASE NOTE FROM 1ST APRIL OUR ADDRESS WILL BE: 55 ORMSIDE WAY, HOLMTHORPE INDUSTRIAL ESTATE, REDHILL, SURREY. TELEPHONE: REDHILL 65080.

From Page 107

at another position in the character row.

Here is a check routine for moving a screen location up by one pixel.

```
.up_pixel       LDA screenlow
                AND #7
                BNE not_at_top
                LDA screenlow
                SEC
                SBC #&79
                STA screenlow
                LDA screenhigh
                SBC #2
                STA screenhigh
                JMP rest_of_program
.not_at_top     DEC screenlow
.rest_of_program
```

Program V moves a byte up the screen one pixel at a time – the reverse of Program IV. Press a key to continue the movement.

It's rather complex isn't it? Wait until you've got to do this for a complete character!

That's enough for this month. Try and absorb as much of it as you can before the next instalment.

```
10 REM PROGRAM V
20 REM By Kevin Edwards
30 MODE 2
40 FOR pass=0 TO 2 STEP 2
50 P%=&B00
60 [OPT pass
70 .start LDA#63
80 .display STA&7D87
90 LDA display+1
100 AND#7
110 BEQ top_of_column
120 DEC display+1
130 RTS
140 .top_of_column LDA display+1
150 SEC
160 SBC#&79
170 STA display+1
180 LDA display+2
190 SBC#2
200 STA display+2
210 RTS
220 ]NEXT
230 FOR loop=1 TO 100
240 CALL start
250 REPEAT UNTIL GET
260 NEXT
```

Line No.	
10-60	Enter assembler.
70	Load accumulator with byte to be stored in video RAM.
80	Store accumulator in video RAM – display byte.
90-100	Check if location of byte displayed is at top of a character row.
110	Branch if it is at top of a character row.
120	Decrement the screen location – move byte up one pixel.
130	Exit.
140-200	Subtract &279 from screen location – this moves the byte to bottom of previous character row.
210	Exit.
220	Exit assembler.
230	Start of loop.
240	CALL the routine.
250	Wait until a key is pressed.
260	End of loop.

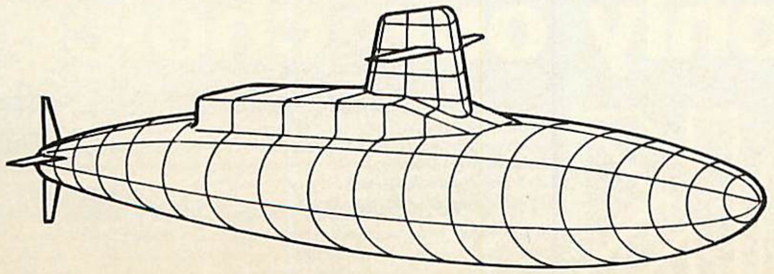
Description of Program V

sub strike

Your submarine is being attacked by wave after wave of enemy helicopters armed with bombs, depth charges and ASGROC (Anti-Submarine Guided ROCkets), the only way to survive is to blast them out of the sky before they can get you - but look out for crashing helicopters, drifting mines and Sonar Subs - and even the fish are dangerous ...

A game of skill, speed and sheer nerve against rapidly increasing odds, from Tomorrows Dream Software.

Fast 100% Machine Code action utilising full colour Mode 2 graphics, sound effects & music, joystick option, High Score table and demo mode, only £6.50 direct from TDS.



TITAN

The debugging monitor utility for machine code programs on the BBC B.

* Single stepping through RAM and ROM - including operating system
* Intelligent memory mover
* Single step through graphics routines
* Breakpoints in RAM and ROM
* Microsecond timer
* Intelligent masked string search routine

Plus:- 'Front Panel' display, OSCLI commands available within debugger, conditional breakpoints at relative branch instructions, display of four memory windows continuosly updated during single stepping, arithmetic operators + – x ÷ , output to printer, ASCII dump, hex dump, dis-assembly, block fill, plus much more.

With complete instruction manual, only £10 on tape, £12 on disc from TDS

Written any good software?
We pay top rates for original programs.

Please send me Sub Strike on tape (£6.50)/disc (£8.50)
TITAN on tape (£10)/disc(£12)
Please specify DFS when ordering disc versions.

I enclose a cheque/PO for £ payable to Tomorrows Dream Software.

Name . . . .

Address (BLOCK LETTERS). . . . .

. . . .

. . . .

Send this coupon with your cheque/PO to: TDS, Richmond House, 1B Sydenham Road, Cotham, Bristol BS6 5SH

Summon up some spritely characters

THIS month we're going to look at software sprites in Mode 2. Sprites are large characters (sometimes multi-coloured) which can be moved around the screen by very simple commands.

They are normally controlled by hardware, but with clever programming techniques they can be simulated with software routines.

All the user has to define is the size of the sprite, its position and the data that corresponds to its shape and colour.

Hardware sprites offer automatic collision detections and have a priority system. They also take up virtually no processing time – in other words, they're very fast. And for most applications where they're used – such as games – speed is essential.

Since sprites are separate from the normal screen display, no background data is lost when they pass over other objects.

They don't even destroy their own kind.

The BBC Micro, unfortunately, does not have any hardware sprites. Programmers have to imitate them using software. The result can never be as good as the real thing, but is suitable for most purposes.

The main difference is speed. Software sprites take a lot of processing time in updating the video RAM. Another problem is that the display may flicker, depending upon when the sprite is displayed and how large it is.

To explain this further we must look at the way television sets and monitors display pictures.

Every fiftieth of a second the screen display is refreshed (updated). The beam of electrons that creates the picture, called the raster, scans across the screen displaying the picture from left to right, working its way down the screen.

When it reaches the bottom right corner it moves back to the top left and starts all over again – one fiftieth of a second later.

The time taken to move from the bottom right to the top left is known as the re-trace period.

The best time to reposition a software sprite is during the re-trace period, when the screen is not being updated. This is easier said than done because the re-trace period is very short.

If you update a software sprite at the same time as the raster displays it the screen will flicker.

For example, if you're half way through updating a sprite and the raster reaches the point on the screen where the sprite is to be displayed, only half of it will appear – the other half hasn't been finished yet.

If this is repeated 50 times a second a horrible flashing character will result – I'm sure you've seen examples of this.

Fortunately we can wait for the start of the re-trace period by issuing a *FX 19 command (OSBYTE 19).

The usual procedure used to update a software sprite is to execute a *FX 19 command, erase the sprite and redisplay it at its new position. This will reduce the flicker. In fact, the display and delete routines are the same. This is because EOR has been used to display the sprites – see the March issue of *The Micro User* for more details about EOR.

If, however, a large sprite is being moved near the top of the screen, flickering will occur even if the above

```
   10 REM Column poke routine
   20 REM By Kevin Edwards
   30 screen=&3100
   40 data=&8000
   60 height=128
   70 MODE 2
   80 HIMEM=&2E00
   90 FORL=0TO2STEP2:P%=HIMEM
  100 [OPTL
  110 .start LDA#height:STA&75
  130 LDA#data MOD 256:STA&72
  140 LDA#data DIV 256:STA&73
  150 LDA#screen MOD 256:STA&76
  160 LDA#screen DIV 256:STA&77
  170 .main_part LDA&76:STA&70
  180 LDA&77:STA&71
  190 LDY#0:LDX&75
  200 .column LDA(&70),Y:EOR(&72),Y
  210 .onto_screen STA(&70),Y
  220 .increment_data INC&72:BNE no_h
igh
  230 INC&73
  240 .no_high LDA&70:AND#7
  250 CMP#7:BNE not_at_bottom
  260 CLC:LDA&70:ADC#&79:STA&70
  270 LDA&71:ADC#2:STA&71
  280 JMP check_end_of_column
  290 .not_at_bottom INC&70
  300 .check_end_of_column DEX:BNE co
lumn
  340 RTS
  350 ]NEXT
  360 CALL start
```

Program I

Line No.	
30	Defines the screen address for the top of the column.
40	Defines the start address for the data which is stored in the column.
60	Defines the height of the column (in pixels).
80-100	Enter the assembler and assemble the code at &2E00.
110-160	Store the parameters in zero page.
170-180	Copy the column start address from &76,&77 into &70,&71 – this is used in later programs.
190	Load the X register with the column height then load the offset register Y (for the indirect addressing) with 0 – the offset is always left at 0.
200-210	Read the screen byte, EOR it with the data byte and store it back in the video RAM.
220-230	Increment the data pointer by one.
240-290	Move the screen pointer down one pixel – see last month's article.
300	Decrement the height register, X, and repeat the previous steps if the column isn't complete.
340	Back to Basic.
350	Exit assembler.
360	Automatically tests the routine.

Description of Program I

Part four
HOW TO WRITE MACHINE CODE GAMES
By KEVIN EDWARDS

procedure is followed. This will be because the raster has completed the retrace and is now displaying the top of the new screen where the large sprite is still being updated – it takes longer to display a large sprite than a small one.

One way to avoid this is to execute *FX 19, wait a while, and then update the sprite. The time delay should be long enough to allow the raster to pass the base of the sprite. Now it is safe to update the screen memory.

As you can see, it's very awkward to produce fast software sprites without flicker. Consider the case where you have 10 sprites moving around the screen. If you used *FX 19 before you updated each one the program would slow down. So the only way to speed it up is to miss a few *FX 19s. The obvious result is flicker. You have to trade off speed versus flicker.

Now we've got that off our chests we'll get down to implementing sprites on the BBC Micro.

Last month we saw how a byte could be moved vertically. One program moved the byte up the screen and the other moved it down. In this article we are only interested in moving a byte down the screen.

The first program, Program I, allows a set of data to be displayed vertically down the screen. The method used to move to the next byte down the screen was described last month.

So far we've never used indirect addressing. Program I uses post-indexed indirect addressing to read a byte from the video RAM, EOR it with the data byte and write it to the screen again. Remember, EORing the screen and data bytes allows us to use one routine for displaying and deleting a sprite.

Indirect addressing works like absolute addressing except that the action address is given by the contents of two consecutive zero page locations which are specified.

The apparently "missing" lines in Program I have been filled in Program II, which allows a sprite of any size to be displayed anywhere on the screen.

The routine starts at the top left corner of a sprite and works its way down, one pixel at a time. On its way it copies a data byte into the screen address being accessed, thus displaying part of the sprite.

When it reaches the base of the sprite it starts again at the top, this time to the right of the previous start address.

Remembering the top location of each column and adding 8 to it provides a simple method of finding the start address for the new column.

```
   10 REM Simple, slow sprite routine
   20 REM By Kevin Edwards
   30 screen=&3100
   40 data=&8000
   50 width=10
   60 height=64
   70 MODE 2
   80 HIMEM=&2E00
   90 FORL=0TO2STEP2:P%=HIMEM
  100 [OPTL
  110 .start LDA#height:STA&75
  120 LDA#width:STA&74
  130 LDA#data MOD 256:STA&72
  140 LDA#data DIV 256:STA&73
  150 LDA#screen MOD 256:STA&76
  160 LDA#screen DIV 256:STA&77
  170 .main_part LDA&76:STA&70
  180 LDA&77:STA&71
  190 LDY#0:LDX&75
  200 .column LDA(&70),Y:EOR(&72),Y
  210 .onto_screen STA(&70),Y
  220 .increment_data INC&72:BNE no_h
igh
  230 INC&73
  240 .no_high LDA&70:AND#7
  250 CMP#7:BNE not_at_bottom
  260 CLC:LDA&70:ADC#&79:STA&70
  270 LDA&71:ADC#2:STA&71
  280 JMP check_end_of_column
  290 .not_at_bottom INC&70
  300 .check_end_of_column DEX:BNE co
lumn
  310 CLC:LDA&76:ADC#8:STA&76:BNE tes
t_end_column
  320 INC&77
  330 .test_end_column DEC&74:BNE mai
n_part
  340 RTS
  350 ]NEXT
  360 CALL start
```
Program II

Program II is the same as Program I with the following additional lines:

50	Defines the width of the sprite (in paired pixels).
120	Copies the sprite's width into zero page.
310-320	Add 8 to the column's previous start address – this moves the column right by two pixels.
330	Tests to see if all of the columns have been displayed. Branch if they haven't.

Description of Program II

In other words, the column routine, Program I is repeated over and over again, with the start address 8 bytes further on in memory each time – two pixels to the right.

This means that the data for the sprite must be stored sequentially column by column. A data pointer keeps track of the data item being displayed – locations &72 and &73 point to the next byte to be displayed.

The dimensions of the sprite are defined in lines 50 and 60. The width corresponds to paired pixels, whereas the height corresponds to single pixels – remember each byte is two pixels wide and one pixel high.

Line 30 specifies the screen address for the top left corner of the sprite. This must be between &3000 and &7FFF – the start and end of the video RAM in Mode 2.

Line 40 defines the start address of the sprite data. This data corresponds to the shape and colour of the sprite.

Programs I, II and III display data from &8000 onwards. &8000 is the start of the Basic ROM and so produces random shapes and colours on the screen.

Program III uses the same techniques

→

HOW TO WRITE MACHINE CODE GAMES

From Page 91

as Program II but in a more efficient way. The speed increase is about 30 per cent.

Software sprite routines tend to vary considerably, depending on the size and movement of the sprites used. Very fast routines are usually tailor made for the sprites being used. The sprite routines given (Programs II and III) are purely for general use.

The aim of them all is to be as fast and efficient on memory as possible. One way to speed them up is to locate the routine in zero page.

The 6502 can execute zero page programs quicker because addressing modes for zero page are quicker than absolute addressing. For example, INC &4000 takes longer to execute than INC &70. Although the time difference is small, when executed several hundred or thousand times it becomes significant.

Program IV uses a faster sprite routine to display a yellow face at the top of the screen.

Lines 50 and 60 inform the routine to display a sprite the same size as a character – 4 bytes by 8.

The problem with the sprite routines we've developed up to now is that they're not very user-friendly when it comes to specifying the position of the sprite.

At the moment you specify the screen location for the top left corner of the sprite. It would be much easier if you could specify the X and Y coordinates of the sprite and let a routine calculate the screen address for you.

The X axis consists of 80 columns

→

```
   10 REM Faster sprite routine
   20 REM By Kevin Edwards
   30 screen=&3100
   40 data=&8000
   50 width=20
   60 height=48
   70 MODE 2
   80 HIMEM=&2E00
   90 FORL=0TO2STEP2:P%=HIMEM
  100 [OPTL
  110 .start LDA#height:STA&75:STA&78
  120 LDA#width:STA&74
  130 LDA#data MOD 256:STAcolumn+1
  140 LDA#data DIV 256:STAcolumn+2
  150 LDA#screen MOD 256:STA&76
  160 LDA#screen DIV 256:STA&77
  170 .user_entry LDX#0
  180 .main_part LDA&76:AND#&F8:STA&7
0
  190 LDA&77:STA&71
  200 LDA&76:AND#7:TAY
  210 .column LDA&FFFF,X:EOR(&70),Y
  220 .onto_screen STA(&70),Y
  230 INX:BEQ inc_data_high
  240 .end_checks INY:CPX&78:BEQ end_
of_column
  250 CPY#8:BNE column
  260 LDA&70:ADC#&7F:STA&70
  270 LDA&71:ADC#2:STA&71
  280 LDY#0:BEQ column
  290 .inc_data_high INC column+2:JMP
 end_checks
  300 .end_of_column CLC:LDA&76:ADC#8
:STA&76:BCC no_high:INC&77
  310 .no_high CLC:LDA&78:ADC&75:STA&
78
  320 DEC&74:BNE main part
  330 RTS
  340 ]NEXT
  350 CALL start
```

Program III

Line No.	
30-60	Define the sprite parameters.
80-100	Enter assembler.
110-160	Copy sprite parameters into other parts of memory. Two of the bytes are stored within the program itself, and the others are put in zero page.
170	Data offset register = 0.
180-190	Copy column start address into &70,&71. The low byte being the top of the current character column (AND &F8).
200	Loads the offset register with the screen low byte ANDed with 7 – to get the offset from the top of the character column.
210-220	Read the data byte, EOR it with the screen byte and store it back in the video RAM.
230	Increments the data offset register. If the X register is zero, branch, so that the data pointer high byte can be incremented.
240	Increments the screen memory address offset register. Check if the end of the column has been reached. If it has, the branch is taken.
250	Checks to see if the start of the next character row column has been reached. Branch if it hasn't.
260-280	Add &280 to the screen address – to move down to the next character row.
290	Increments the data pointer high byte and continues with the rest of the column.
300	Adds eight to the column's start address – the top of it. This moves the start address right by two pixels.
310	Changes the data offset stop pointer so that the end of the next column will be terminated correctly.
320	Checks to see if all of the columns have been displayed. Branch if they haven't.
330	Exits to Basic.
340	Exits assembler.
350	Tests program.

Description of Program III

```
   10 REM Faster sprite routine
   20 REM By Kevin Edwards
   30 screen=&3100
   40 data=&C00
   50 width=4
   60 height=8
   70 MODE 2
   80 HIMEM=&2E00
   90 FORL=0TO2STEP2:P%=HIMEM
  100 [OPTL
  110 .start LDA#height:STA&75:STA&78
  120 LDA#width:STA&74
  130 LDA#data MOD 256:STAcolumn+1
  140 LDA#data DIV 256:STAcolumn+2
  150 LDA#screen MOD 256:STA&76
  160 LDA#screen DIV 256:STA&77
  170 .user_entry LDX#0
  180 .main_part LDA&76:AND#&F8:STA&7
0
  190 LDA&77:STA&71
  200 LDA&76:AND#7:TAY
  210 .column LDA&FFFF,X:EOR(&70),Y
  220 .onto_screen STA(&70),Y
  230 INX:BEQ inc_data_high
  240 .end_checks INY:CPX&78:BEQ end
of_column
  250 CPY#8:BNE column
  260 LDA&70:ADC#&7F:STA&70
  270 LDA&71:ADC#2:STA&71
  280 LDY#0:BEQ column
  290 .inc_data_high INC column+2:JMP
 end_checks
  300 .end_of_column CLC:LDA&76:ADC#8
:STA&76:BCC no_high:INC&77
  310 .no_high CLC:LDA&78:ADC&75:STA&
78
  320 DEC&74:BNE main_part
  330 RTS
  340 ]NEXT
  350 FORL%=0 TO 31
  360 READ L%?data
  370 NEXT
  380 CALL start
  390 DATA5,15,10,15,15,15,15,5
  400 DATA15,15,5,15,15,5,0,15
  410 DATA15,15,10,15,15,10,0,15
  420 DATA10,15,5,15,15,15,15,10
```

Program IV

Program IV is the same as Program III with the following additions or changes:

30-60	The sprite parameters have been changed to display a 8 by 4 byte sprite at location &3100 on the screen – the data starts at &C00.
350-370	Read and store the sprite data into page &C.
390-420	The sprite data – a smiling face.

Description of Program IV

Memory map for the sprite routines:

&70,&71	Screen address pointer.
&72,&73	Data pointer.
&74	Width of sprite, decremented each time a column is displayed.
&75	Height of the sprite in pixels.
&76,&77	Copy of the column start address – the top of the current column.

```
   10 REM Screen location calculator.
   20 REM Using screen X and Y co-ord
s.
   30 REM Origin - top left corner.
   40 REM Uses O.S for data tables.
   50 REM By Kevin Edwards
   60 MODE7:HIMEM=&7000
   70 FORL=0TO2STEP2:P%=HIMEM
   80 [OPTL
   90 .calc_loc LDA#&30:STA&81
  100 LDA#0:STA&82
  110 TYA:AND#7:STA&80
  120 TYA:LSRA:LSRA:LSRA:ASLA:TAY
  130 TXA:ASLA:ROL&82:ASLA:ROL&82
  140 ASLA:ROL&82
  150 ADC&80:ADC&C376,Y:STA&80
  160 LDA&82:ADC&81:ADC&C375,Y:STA&81
  170 RTS
  180 ]NEXT
  190 REPEAT
  200 INPUT''"Enter X co-ordinate",X%
  210 IF X%<0 OR X%>79 THEN VDU7:PRIN
T'"X must be in the range 0-79 !":GOT
O 200
  220 INPUT'"Enter Y co-ordinate",Y%
  230 IF Y%<0 OR Y%>255 THEN VDU7:PRI
NT'"Y must be in the range 0-255 !":G
OTO 220
  240 CALL calc_loc
  250 PRINT'"Location =&";~?&81*256+?
&80
  260 UNTIL 1=2
```

Program V

Line No.	
60-80	Enter assembler.
90	Stores Mode 2 screen start address high byte in &81.
100	Stores 0 in the workspace. This is used to accommodate any overflow that occurs when X is multiplied by 8.
110	Puts Y AND 7 into the screen address low byte.
120	Calculates (Y/8)*2. The result of Y/8 is multiplied by 2 to allow us to make use of the ROM multiplication table – this starts at &C375 and consists of 32 entries of an &280 multiplication table. Each entry in the table is stored in pairs – high byte then low byte.
130-140	Calculate X*8. The three most significant bits of the result are held in bits 0 to 2 of location &82 the workspace.
150	Adds the screen low byte to the result of X*8. Then adds the low byte of the &280 multiplication table to the result. This is then stored in location &80.
160	Adds the high byte of X*8 to the screen address high byte. Then adds the high byte of the &280 multiplication table and stores the result in location &81.
170	Back to Basic.
180	Exit assembler.
190-260	Accept valid input for two screen co-ordinates and print the result.

Description of Program V

HOW TO WRITE MACHINE CODE GAMES

From Page 93

(0-79) and the Y axis has 256 rows (0-255) – see Figure I, page 74 in the February issue of *The Micro User*. If we use the top left corner as the origin we can use the following equation to calculate the screen location.

```
location=&3000+(X*8)+&280*(Y DIV 8)+(Y
AND 7)
```

As you can see, it's quite a complex equation. The machine code equivalent is given in Program V.

When you RUN the program you'll be asked to enter the X and Y coordinates of the sprite's position – remember the origin is the top left corner of the screen. On entering these the screen address will be calculated and displayed.

On entry to the routine the X register holds the X coordinate of the sprite and the Y register contains its Y coordinate. These are passed to the routine by the resident integer variables X% and Y%.

So in your own machine code programs you would load the X and Y registers with the X and Y coordinates of the sprite and then JSRcalc_loc.

On exit, location &80 contains the low byte of the screen address and &81 contains the high byte of the screen address. It's as easy as that.

Program VI contains the screen location calculator and the sprite routine from Program III. It demonstrates how the two routines can be combined to move a multi-coloured sprite across the screen.

Now you've got two sprite routines I'm sure you'll be able to conjure up some stunning animation.

Next month's article will contain a full listing of a multi-coloured character definer. Until then, happy animation.

```
   10 REM Sprite demo
   20 REM By Kevin Edwards
   30 data=&C00
   40 width=4
   50 height=16
   60 MODE 2
   70 HIMEM=&2E00
   80 FORL=0TO2STEP2:P%=HIMEM
   90 [OPTL
  100 .sprite LDA#height:STA&75:STA&7
8
  110 LDA#width:STA&74
  120 LDA#data MOD 256:STAcolumn+1
  130 LDA#data DIV 256:STAcolumn+2
  140 .user_entry LDX#0
  150 .main_part LDA&76:AND#&F8:STA&7
0
  160 LDA&77:STA&71
  170 LDA&76:AND#7:TAY
  180 .column LDA&FFFF,X:EOR(&70),Y
  190 .onto_screen STA(&70),Y
  200 INX:BEQ inc_data_high
  210 .end_checks INY:CPX&78:BEQ end_
of_column
  220 CPY#8:BNE column
  230 LDA&70:ADC#&7F:STA&70
  240 LDA&71:ADC#2:STA&71
  250 LDY#0:BEQ column
  260 .inc_data_high INC column+2:JMP
 end_checks
  270 .end_of_column CLC:LDA&76:ADC#8
:STA&76:BCC no_high:INC&77
  280 .no_high CLC:LDA&78:ADC&75:STA&
78
  290 DEC&74:BNE main_part
  300 RTS
  310 .calc_loc LDA#&30:STA&81
  320 LDA#0:STA&82
  330 TYA:EOR#&FF:TAY:AND#7:STA&80
  340 TYA:LSRA:LSRA:LSRA:ASLA:TAY
  350 TXA:ASLA:ROL&82:ASLA:ROL&82
  360 ASLA:ROL&82
  370 ADC&80:ADC&C376,Y:STA&76
  380 LDA&82:ADC&81:ADC&C375,Y:STA&77
  390 RTS
  400 ]NEXT
  410 FORL%=0 TO 63
  420 READ L%?data
  430 NEXT
  440 Y%=100
  450 FORX%=0 TO 75
  460 CALLcalc_loc
  470 CALLsprite
  480 *FX19
  490 CALLcalc_loc
  500 CALLsprite
  510 Y%=Y%+2*(RND(3)-2)
  520 NEXT
  530 END
  540 REM Monster data
  550 DATA&3F,0,1,1,9,9,9,9
  560 DATA9,10,10,0,0,0,32,&33
  570 DATA0,&2A,3,6,&24,&24,12,4
  580 DATA8,12,12,8,8,8,8,&22
  590 DATA0,&15,3,9,&18,&18,12,8
  600 DATA4,12,12,4,4,4,4,&11
  610 DATA&3F,0,2,2,6,6,6,6
  620 DATA6,5,5,0,0,0,&10,&33
```

Program VI

Line No.	
30-50	Define the sprite's parameters.
70-90	Enter the assembler.
100-300	The main sprite routine – see Program III's description.
310-390	Screen location calculation routine. On entry X and Y contain the screen X and Y coordinates. In this example the origin is at the bottom left of the screen. Line 330 EORs the Y coordinate with &FF to change the origin from the top left corner to the bottom left corner.
410-430	Read and store the sprite's graphic data into page &C.
440	Defines the sprite's Y coordinate.
450	Causes the sprite's X coordinate to take all of the values between 0 and 75.
460-470	Calculate the screen address for the sprite and then display it.
480	Waits for the start of the retrace period.
490-500	Delete the old sprite.
510	Randomly changes the Y coordinate of the sprite.
520	Next X coordinate.
540-620	Sprite data.

Description of Program VI

Part five

HOW TO WRITE MACHINE CODE GAMES

By KEVIN EDWARDS

It's a sort of multicoloured swap shop

THIS month we're going to look at two more sprite routines and we'll also see how useful a multicoloured character definer can be.

Program I is the Mode 2 multi-coloured character definer as promised last month. This will allow you to design your own sprite graphics with minimal effort. These sprites can then be displayed on the screen by using the other programs given in this article.

Sprites, for the uninformed, are multicoloured characters that can be freely moved around the screen without destroying any background information.

Until now the only way to design sprites was to use graph paper and coloured pens. The problem is that this technique is very time consuming. So to save time and trouble I have designed a simple sprite definer. Type it in, save it and then RUN.

The first question you'll be asked is "Load old file Y/N?" At this point your reply should be N – you haven't created a data file yet!

If you have saved a data file you can load it back in by pressing Y, after which the file name will be requested. On entering this the file is loaded and the editing mode is entered.

Assuming you have not just loaded a file, you will be asked for the sprite's dimensions. These will be provided by data held at the start of the file. (Skip the next few paragraphs if you've loaded a sprite from a file.)

Now you will be asked for the width of your sprite in paired pixels – between 1 and 8. Remember that each byte of the Mode 2 screen corresponds to two pixels.

Predictably, the next question asks you for the height of the sprite – this time in multiples of eight pixels (the height of one character cell).

The editing mode will be entered as soon as the sprite's height has been typed in. If you've loaded a file from tape or disc, you'll be in the editing mode already.

A grid, directly corresponding to the size of the sprite, will be displayed on the screen. Underneath it is the colour palette which consists of 16 differently coloured blocks.

Below the red block is the colour pointer. This indicates the colour of the editing pen – that is, the colour you'll be filling the cells of the grid in with. Initially the editing pen is red.

The pointer, and hence the colour, can be changed by pressing the f0 and f1 keys. f0 selects the colour to the left of the colour pointer and f1 selects the colour to its right.

You should have also noticed the cursor flashing at grid reference A,A. This cursor is your means of indicating which pixel of the sprite is being edited.

You can move the cursor anywhere in the grid by pressing the appropriate cursor arrow key. Once you've correctly positioned the cursor you can change the colour of the cell to the selected pen colour by pressing Space. Doing this when the pen is black clears the cell, since black is the background colour.

When being edited the sprite is magnified several times. To view the sprite "life-size" press f2. It will now be displayed to the right of the editing grid. The editing mode will be automatically resumed.

If at any time you want to fill the whole grid with the palette colour you can do so simply by pressing f8. Be very careful though – your previous design will be erased.

If you wish to redefine the sprite's dimensions you press f7 while in the editing mode, which has the effect of rerunning the program.

When you are satisfied with your creation you can save the sprite to tape or disc by pressing S.

You will then be asked whether or not you wish to save just the sprite data. Entering N indicates to the program that the sprite's dimensions should also be saved. This information is needed if the sprite is to be reloaded at another time for editing.

Pressing Y saves just the sprite data. This means that the sprite dimensions are not saved and that the file cannot be loaded by the definer for editing.

The data is saved in this way so that the sprite routines, introduced in last month's article, can be used to display the data. In other words, the data is saved column by column from the left of the sprite to the right.

After either selection you will be asked for the file name under which the file will be saved. This must be a maximum of seven characters in length so that the program can be used with disc systems.

The final command, L, allows you to load a new sprite into memory.

After pressing L the file name of the sprite should be entered. The sprite will now be loaded – make sure you've saved the previous sprite beforehand or you will lose it forever.

Also remember that the file cannot be loaded if the sprite's dimensions were

→

```
   10 REM Multi-size Sprite routine
   20 REM By Kevin Edwards
   30 MODE 2
   40 HIMEM=&2E00
   50 FORL=0TO2STEP2:P%=HIMEM
   60 [OPTL
   70 .sprite STX&74
   80 STY&75:STY&78
   90 .user_entry LDX#0
  100 .main_part LDA&76:AND#&F8:STA&7
0
  110 LDA&77:STA&71
  120 LDA&76:AND#7:TAY
  130 .column LDA&FFFF,X:EOR(&70),Y
  140 .onto_screen STA(&70),Y
  150 INX:BEQ inc data_high
  160 .end_checks INY:CPX&78:BEQ end_
of_column
  170 CPY#8:BNE column
  180 LDA&70:ADC#&7F:STA&70
  190 LDA&71:ADC#2:STA&71
  200 LDY#0:BEQ column
  210 .inc_data_high INC column+2:JMP
 end_checks
  220 .end_of_column CLC:LDA&76:ADC#8
:STA&76:BCC no_high:INC&77
  230 .no_high CLC:LDA&78:ADC&75:STA&
78
  240 DEC&74:BNE main_part
  250 RTS
  260 .calc_loc LDA#&30:STA&81
  270 LDA#0:STA&82
  280 TYA:EOR#&FF:TAY:AND#7:STA&80
  290 TYA:LSRA:LSRA:LSRA:ASLA:TAY
  300 TXA:ASLA:ROL&82:ASLA:ROL&82
  310 ASLA:ROL&82
  320 ADC&80:ADC&C376,Y:STA&76
  330 LDA&82:ADC&81:ADC&C375,Y:STA&77
  340 RTS
  350 ]NEXT
  360 FOR L%=0 TO &5F
  370 READ L%?&C00
  380 NEXT
  390 FOR X1%=0 TO 70
  400 PROCsprite_onoff
  410 *FX 19
  420 PROCsprite_onoff
  430 NEXT X1%
  440 END
  450 DEFPROCsprite_onoff
  460 X%=X1%:Y%=255:CALLcalc_loc
  470 column?1=0:column?2=&C
  480 X%=4:Y%=24:CALLsprite
  490 ENDPROC
  500 DATA 0,0,0,&11,&11,0,&11,&11
  510 DATA 0,0,0,0,0,0,3,3
  520 DATA 1,1,1,1,0,0,0,0
  530 DATA 0,&11,&11,&33,&27,&27,&27,
&33
  540 DATA &11,&11,4,4,4,4,3,&2B
  550 DATA &2B,&2B,&2B,&2B,&17,&17,&1
7,&17
  560 DATA 0,&22,&22,&33,&1B,&1B,&1B,
&33
  570 DATA &22,&22,8,8,8,8,3,3
  580 DATA 3,3,3,3,3,3,3,3
  590 DATA 0,0,0,&22,&22,0,&22,&22
  600 DATA 0,0,0,0,0,0,3,3
  610 DATA 2,2,2,2,0,0,0,0
```

Program II

From Page 91

not saved when the file was originally created.

Files which contain only sprite data are ready to be used with the sprite display routine without modification. All you need to do is *LOAD the data into page &C (or wherever the data is needed) and change the sprite parameters in the sprite routine accordingly.

The start of your sprite program should look something like this:

```
10 *LOAD SPRITE1 C00
20 width=4
30 height=16
40 data=&C00
50 REM Start of sprite routine
```

All this is quite simple so far. What we need now is a routine that will be capable of displaying sprites of varying sizes after the program has been assembled.

After all, I'm sure you'll be using sprites of different sizes in your own programs. A routine capable of displaying only one size of sprite is rather limiting.

Ideally, what we require is a routine we can call with the dimensions of the sprite held in the 6502 X and Y registers. In fact, only a few changes are

HOW TO WRITE MACHINE CODE GAMES

required to last month's sprite routine to do this.

Program II is the modified sprite routine along with the screen location calculator from last month's article. This is a clever routine that calculates the screen location, for Mode 2, of a specified X and Y coordinate using the bottom left corner as the origin.

Program II shows how easy it is to move a sprite – in this case a flower pot – across the screen using the new routine.

To display a sprite using the new routine all we need to do is:

- Call the screen location calculator routine (.calc_loc) with the screen X and Y coordinates in the 6502 X and Y registers – line 460 of Program II. The result is automatically stored in zero page, ready for the sprite routine.
- The pointer to the sprite graphic data must be stored into locations (column+1) and (column+2) – low byte then high. See line 470 of Program II.
- The dimensions of the sprite should be loaded into the X and Y registers.

Now the main sprite routine can be called (.sprite) – line 480 of Program II. This displays/erases the sprite on/from the screen. Remember, the sprite routine uses EOR!

It's still rather long winded, but there's no simple way of passing so many parameters to the relevant routines.

The effects produced by all the programs so far have relied on a little bit of Basic. The problem with this is that jumping in and out of Basic takes up valuable time.

This is made even worse when sections of a Basic program are executed in between machine code routines – especially when timing is critical such as when using *FX 19 to wait for start of retrace period.

So now we move on to Program III,

Line No.	
40-60	Enter assembler and assembles code at &2E00.
70	Stores width of sprite in &74 – in paired pixels.
80	Stores height of sprite in &75 and &78 – in pixels.
90	Data offset register = 0.
100-110	Copy column start address into &70, &71, the low byte being the top of the current character column (AND &F8).
120	Loads offset register with screen low byte ANDed with 7 – to get the offset from the top of the character column.
130-140	Read data byte, EOR it with screen byte and store it back in video RAM.
150	Increments data offset register. If X register is zero, branch so that data pointer high byte can be incremented.
160	Increments screen memory address offset register. Checks if end of column has been reached. If it has, the branch is taken.
170	Checks to see if start of next character row column has been reached. Branches if it hasn't.
180-200	Add &280 to screen address to move down to next character row. Then change screen offset register, Y, to 0.
210	Increments data pointer high byte and continues with rest of the column.
220	Adds eight to column's start address. This moves start address right by two pixels.
230	Changes data offset stop pointer so that end of next column will be terminated correctly.
240	Checks to see if all of the columns have been displayed. Branches if they haven't.
250	Exits from sprite routine.
260	Stores Mode 2 screen start address high byte in &81.
270	Stores 0 in the workspace. This is used to accommodate any overflow that occurs when X is multiplied by 8.
280	EORs the Y coordinate with &FF (to change origin from top left corner to bottom left corner) and then ANDs result with 7. This tells us how many whole character rows down the screen the sprite is.
290	Calculates INT(Y/8)*2. The result of Y/8 is multiplied by 2 to allow us to make use of ROM multiplication table. This starts at &C375 and consists of 32 entries of an &280 multiplication table. Each entry in table is stored in pairs – high byte then low byte.
300-310	Calculates X*8. The three most significant bits of the result are held in bits 0 to 2 of location &82 the workspace.
320	Adds screen low byte to low byte of result of X*8. Then adds low byte of &280 multiplication table to result. The final total is stored in location &76.
330	Adds high byte of X*8 to screen address high byte. Then adds high byte of &280 multiplication table and stores result in location &77 – any carry which may have occurred from the addition of the low bytes is also included in the result.
340	Back to Basic.
350	Exits assembler.
360-380	Read and store sprite data – a flower pot.
390	Start of a loop – this defines X coordinate of sprite.
400	Displays sprite on screen.
410	Waits for retrace period.
420	Erases sprite from screen.
430	Next X coordinate.
440	End of program.
450-490	The sprite display/erase procedure.
460	Sets X% and Y% to sprite's screen coordinates and calls screen location calculator routine.
470	Resets sprite data pointer.
480	Defines sprite's dimensions before CALLing main sprite routine which in turn displays/erases the sprite.
500-610	Flower pot data.

Description of Program II

which uses only machine code to move the flower pot up the screen.

The procedure PROCsprite_onoff, from Program II, has been replaced by a subroutine (.onoff) – lines 100-120.

Location &83 is used to keep track of the sprite's Y coordinate. Between each movement, 4 is added to Y to move the sprite up 4 pixels – line 90. Try changing line 90 so the sprite moves one pixel at a time. All you need to do is change the ADC#4 to ADC#1.

By using simple programming techniques you'll be able to produce fully animated sprites. This can be achieved by flicking between different sets of sprite data just like flicking through a book which has different sketches on each page.

I'll leave that problem for you to solve – it's not as hard as it sounds.

See you next month when we'll look at some more useful techniques found in machine code games.

```
   10 REM Full machine code demo
   20 REM By Kevin Edwards
   30 MODE 2
   40 HIMEM=&2E00
   50 FORL=0TO2STEP2:P%=HIMEM
   60 [OPTL
   70 .demo LDA#23:STA&83
   80 .demo1 JSRonoff:LDA#19:JSR&FFF4
:JSRonoff
   90 LDA&83:CLC:ADC#4:STA&83:BCCdemo
1:RTS
  100 .onoff LDX#20:LDY&83:JSRcalc_lo
c
  110 LDA#0:STAcolumn+1:LDA#&C:STAcol
umn+2
  120 LDX#4:LDY#24:JSRsprite:RTS
  130 .sprite STX&74
  140 STY&75:STY&78
  150 .user_entry LDX#0
  160 .main_part LDA&76:AND#&F8:STA&7
0
  170 LDA&77:STA&71
  180 LDA&76:AND#7:TAY
  190 .column LDA&FFFF,X:EOR(&70),Y
  200 .onto_screen STA(&70),Y
  210 INX:BEQ inc_data_high
  220 .end_checks INY:CPX&78:BEQ end_
of_column
  230 CPY#8:BNE column
  240 LDA&70:ADC#&7F:STA&70
  250 LDA&71:ADC#2:STA&71
  260 LDY#0:BEQ column
  270 .inc_data_high INC column+2:JMP
 end_checks
  280 .end_of_column CLC:LDA&76:ADC#8
:STA&76:BCC no_high:INC&77
```

→

Program III

Join the communications revolution

We've found the missing link!

By DEREK MEAKIN
Managing Editor The Micro User

MANY months ago we forecast that 1985 would be the Year of Communications. And so it is turning out.

Sales of modems are soaring. Micro users are joining Prestel in their thousands. Bulletin boards are springing up all over the country.

But until now one thing has been missing.

What has been desperately needed was a national database for microcomputers that would combine a low-cost electronic mail service with a giant bulletin board you could access first time every time, that would allow you to chat freely with other users, that would have no restriction on the length of material you could send or receive – and would also enable you to send telex and telemessages from your own micro.

All this, and much much more, you can do with MicroLink. For this one comprehensive service is the missing link in Britain's communications network.

And the carrier through which all MicroLink's facilities are being offered is the most respected name of all – Telecom Gold.

Until now Telecom Gold has been the preserve of big business – used by the giants of industry, commerce and finance to link their operations in Britain with the rest of the world. And Telecom Gold's charges reflected its elevated status.

That has now changed. Thanks to MicroLink, every micro user can afford to key into Telecom Gold and make use of its growing range of services. For less than the cost of a first class stamp you can have a message the size of this page delivered to one destination or hundreds . . . all in a fraction of a second. And that's just one example of the time and money you'll be able to save with MicroLink.

All of us at *The Micro User* are very excited about the enthusiastic reception the new service has been given by our readers – and about its unlimited potential for future development.

Come and join us – and explore with us the whole new world that is being opened up by MicroLink.

HOW TO WRITE MACHINE CODE GAMES

From Page 93

```
  290 .no_high CLC:LDA&78:ADC&75:STA&
78
  300 DEC&74:BNE main_part
  310 RTS
  320 .calc_loc LDA#&30:STA&81
  330 LDA#0:STA&82
  340 TYA:EOR#&FF:TAY:AND#7:STA&80
  350 TYA:LSRA:LSRA:LSRA:ASLA:TAY
  360 TXA:ASLA:ROL&82:ASLA:ROL&82
  370 ASLA:ROL&82
  380 ADC&80:ADC&C376,Y:STA&76
  390 LDA&82:ADC&81:ADC&C375,Y:STA&77
  400 RTS
  410 ]NEXT
  420 FOR L%=0 TO &5F
  430 READ L%?&C00
  440 NEXT
  450 CALLdemo
  460 END
  470 DATA 0,0,0,&11,&11,0,&11,&11
  480 DATA 0,0,0,0,0,0,3,3
  490 DATA 1,1,1,1,0,0,0,0
  500 DATA 0,&11,&11,&33,&27,&27,&27,
&33
  510 DATA &11,&11,4,4,4,4,3,&2B
  520 DATA &2B,&2B,&2B,&2B,&17,&17,&1
7,&17
  530 DATA 0,&22,&22,&33,&1B,&1B,&1B,
&33
  540 DATA &22,&22,8,8,8,8,3,3
  550 DATA 3,3,3,3,3,3,3,3
  560 DATA 0,0,0,&22,&22,0,&22,&22
  570 DATA 0,0,0,0,0,0,3,3
  580 DATA 2,2,2,2,0,0,0,0
```

The machine code routines in Program III are identical to those in Program II with the following additions.

Line No.

70 Stores sprite's Y coordinate in location &83.

80 Displays sprite, waits for the retrace period and then erases sprite. Remember the display and erase routines are the same.

90 Adds 4 to sprite's Y coordinate to move it up the screen by 4 pixels. A branch is taken if sprite's Y coordinate hasn't overshot top of screen. This repeats process for a new position. Otherwise program finishes.

100 Calculates screen location for the sprite's X and Y coordinates.

110 Resets sprite data pointer so that data starting at &C00 is displayed.

120 Defines sprite's dimensions before calling sprite display/erase routine.

Part six

HOW TO WRITE MACHINE CODE GAMES

By KEVIN EDWARDS

How to keep (and raise) your score

NEARLY all games have a score of some kind, and wherever there's a score there's a high score table. These are two of the subjects we'll be looking at this month.

First we'll consider how to keep and increase a score.

The easiest way to store the score is as a sequence of numeric Ascii characters – 0 to 9. This makes it easier to output, as we shall see.

One very useful routine in the operating system is OSWRCH (&FFEE). Calling this routine will print the character whose Ascii code is held in the accumulator – just what we need to print out the score.

For example, if the accumulator contains 65 and OSWRCH is called, the letter A is printed on the screen – 65 being the Ascii code for A. You see, OSWRCH is the machine code equivalent of VDU.

Take a look at the following example:

```
LDA#31
JSR &FFEE
LDA#0
JSR &FFEE
LDA#6
JSR &FFEE
```

This is exactly the same as VDU 31,0,6 – move the text cursor to coordinate 0,6.

OSWRCH allows you to perform many operations from drawing triangles to turning printers on and off. See page 378 of the User Guide for a list of the VDU codes.

Program I uses OSWRCH to position the text cursor (line 170) and print out the score digits (line 190).

The score is held in eight consecutive locations. This means that the largest number that can be represented is 99999999! The base location of the score is pointed to by the variable *thescore*.

Program I contains two routines. One increases the score by 10 points (lines 90-160) and the other prints the score on the screen (lines 170-210).

These have been separated so that the *add 10* subroutine can be called several times to make it into an *add 10*n* subroutine – *n* being the number of times the routine is called.

In other words we can call the *add 10* routine 5 times to add 50 to the score before we print it on the screen.

Some games use score routines which print leading zeros. For example, one game might print the score as "000150" and another would print it as "150". I'm sure that you'll agree that the latter looks the tidier.

So to make the score output neat and tidy leading zeros will be replaced by spaces. This must be taken into account by the score routine.

Try Program I, pressing a key to add 10 to the score.

Line 230 is very important. This initialises the score to 0. The length of the string should always be eight characters in total with a zero at the end.

The score is increased by incrementing the Ascii character in the tens position of the score – remember Hundreds Tens and Units. If the digit in the tens position is a nine it must be replaced by a zero and 1 must be added to the hundreds column.

In fact the same routine which increased the tens is used to add one to the hundreds. This is achieved by decrementing the score offset register so that it points to the hundreds column instead of the tens column – the most significant digits of the score are stored first.

Again, if the digit is a 9 the routine puts a zero in the score, decrements the offset register and branches back on itself. You'll be able to understand this better by following Program I through in your head.

One of the features that has been added to more and more games is the high score table. This allows a person who has accumulated a good score to enter their name into a table of other scores.

Most high score tables keep a record of the top ten scores. The higher the score the higher its position is in the table.

When a new high score is recorded space must be provided to accommodate the new score. This is done by moving all the scores less than the new one down the table one place. This means that the lowest score must be lost.

The name of the person who achieved the new score is requested and entered into the table.

It gives players great satisfaction when they see their name and score above all of the others. It's also very useful when several players are having a competition.

Achieving this in machine code is quite tricky. Several problems must be solved first:

- *Storing the high scores and names in memory.*
- *Accepting the name of the person who achieved the high score.*

The obvious way to store the names and scores is in Ascii. This makes it easier for us to print the table – using OSWRCH – and to use the Ascii characters, corresponding to new score, from Program I.

Program II is the high score table routine which keeps a record of the top

```
  10 REM Score routine
  20 REM By Kevin Edwards
  30 MODE 7
  40 HIMEM=&2800
  50 thescore=&2AA0
  60 FOR pass=0 TO 2 STEP 2
  70 P%=HIMEM
  80 [OPT pass
  90 .add10 LDX#6
 100 .examinedigit LDAthescore,X
 110 CMP#ASC" ":BEQmake_it_a_1
 120 CMP#ASC"9":BNEincdigit
 130 LDA#ASC"0":STAthescore,X
 140 DEX:BPLexaminedigit
 150 .make_it_a_1 LDA#ASC"1":STAthes
core,X:RTS
 160 .incdigit INCthescore,X:RTS
 170 .scoreout LDA#31:JSR&FFEE:LDA#1
5:JSR&FFEE:LDA#12:JSR&FFEE
 180 LDY#0
 190 .nextdig LDAthescore,Y:JSR&FFEE
 200 INY:CPY#8:BNEnextdig
 210 RTS
 220 ]NEXT
 230 $thescore="       0"
 240 REPEAT
 250 CALLscoreout
 260 CALLadd10
 270 A%=GET
 280 UNTIL 1=2
```
Program I

40-80	Enter assembler.
90	Sets score offset register to 6.
100	Reads digit.
110	checks if it's a space. If it is make it a one. Spaces are used instead of leading zeros.
120	Is the digit a 9? Branch if it isn't.
130-140	If it's a space make it a zero and repeat the process for the next digit along.
150	Makes the score digit 1.
160	Increments the score digit by one to add 10 to the score and then exit.
170	Positions the text cursor – VDU 31,15,12.
180-200	Output the score digits.
210	Exits the routine.

Description of Program I

nine scores. Run it and enter your name.

Line 860 specifies the new score. Try changing it, but make sure that the score string is eight characters in total.

The high score routine execution address is pointed to by the variable *start*. Make sure you've set the score up in memory before you call the routine – the score address is given by the variable *newscore*.

The names of the high scores are stored sequentially in nine groups of 16 characters. Only 15 characters of each hold the name, the other holds CHR$(13) – carriage return. The variable *names* points to the first name.

The scores are stored sequentially in nine groups of eight characters. The first character in each group is the most significant digit of the score.

The names and scores were limited to 16 and eight characters to simplify calculations needed to locate a specific score by multiplication. For example, the start of the text for the third score can be found by multiplying 8 (number of characters per score) by the position, 1, the result being 16 (8*(3–1)).

A number can be multiplied by 8 by shifting it left three times using the command ASL – see line 220. Shifting it left once more, again using ASL, is the same as multiplying the original number by 16 – see line 230.

Of course this only works if the original number is less than 16 – which it will be in our case. If it is greater or equal to 16 the result will overflow because the maximum number that a register or location can hold is 255.

The second problem was how to accept an input from the user when his name is requested for the high score table. This is a lot easier than it sounds. Another operating system call allows us to do this.

The call required is to OSWORD (&FFF1) with the accumulator set to 0 – OSWORD 0 for short. The X and Y registers hold the low and high byte, respectively, of a control block which gives further information about the operation to be performed.

For example, if the control block is at location &A60 the following routine would execute OSWORD 0 correctly.

```
LDX#&60
LDY#&A
LDA#0
JSR &FFF1
```

But before this can be executed the control block must be set up. This specifies the following information:

- *The address in memory where the input string is to be stored – low byte then high byte.*
- *The maximum number of characters to be accepted.*
- *The minimum acceptable Ascii character.*
- *The maximum acceptable Ascii character.*

For example, we'll set up the control

→

```
  10 REM High Score Table
  20 REM By Kevin Edwards
  30 MODE 7
  40 HIMEM=&2800
  50 names=&2A00
  60 newscore=&2AA0:workspace=&2AB0
  70 scores=&2B00
  80 FOR pass=0 TO 2 STEP 2
  90 P%=HIMEM
 100 [OPT pass
 110 .start LDA#22:JSR&FFEE:LDA#7:JS
R&FFEE
 120 LDX#0:TXA:PHA
 130 .checkanother ASLA:ASLA:ASLA:TA
X:LDY#0
 140 .char1 LDAnewscore,Y:CMPscores,
X:BCCnotgreater:BNEinsert
 150 INX:INY:CPY#8:BCCchar1
 160 BCSinsert
 170 .notgreater PLA:TAX:INX:TXA:PHA
:CPX#9:BCCcheckanother
 180 LDA#255:STAworkspace+2:JMPfinis
h
 190 .insert LDY#0
 200 .messageout LDAmessage,Y:BEQexi
t:JSR&FFEE:INY:BNEmessageout
 210 .exit PLA:PHA:STAworkspace+2
 220 ASLA:ASLA:ASLA:STAworkspace
 230 ASLA:STAworkspace+1
 240 CMP#128:BEQmovenewscore
 250 LDX#127
 260 .copynames LDAnames,X:STAnames+
16,X:DEX:CPXworkspace+1:BNEcopynames
 270 LDAnames,X:STAnames+16,X
 280 LDX#63
 290 .copyscores LDAscores,X:STAscor
es+8,X:DEX:CPXworkspace:BNEcopyscores
 300 LDAscores,X:STAscores+8,X
 310 .movenewscore LDXworkspace:LDY#
0
 320 .copynewscore LDAnewscore,Y:STA
scores,X:INX:INY:CPY#8:BCCcopynewscor
e
 330 .deleteoldname LDXworkspace+1:L
DY#15:LDA#32
 340 .delete STAnames,X:INX:DEY:BPLd
elete
 350 LDAworkspace+1:STAoswordblock:L
DA#0:LDX#oswordblock MOD 256:LDY#oswo
rdblock DIV 256:JSR&FFF1
 360 .finish PLA:JSRdisplay
 370 LDY#0
 380 .messageout2 LDAmessage2,Y:BEQe
xit2:JSR&FFEE:INY:BNEmessageout2
 390 .exit2 JSR&FFE0:LDA#&7E:JMP&FFF
4
 400 .display LDA#12:JSR&FFEE
 410 .displayscores LDX#0
 420 .nextscore LDA#31:JSR&FFEE:LDA#
1:JSR&FFEE:TXA:ASLA:ADC#3:JSR&FFEE
 430 TXA:AND#7:CMP#7:BNEcolourok:LDA
#2
 440 .colourok CLC:ADC#129:JSR&FFEE
 450 LDA#32:CPXworkspace+2:BNEpositi
on:LDA#136
 460 .position JSR&FFEE
 470 TXA:CLC:ADC#49:JSR&FFEE
 480 LDY#3:JSRdots
 490 TXA:PHA:ASLA:ASLA:ASLA:TAX:LDY#
7
 500 .nextdigit LDAscores,X:JSR&FFEE
:INX:DEY:BPLnextdigit
 510 LDY#4:JSRdots
 520 PLA:PHA
 530 ASLA:ASLA:ASLA:ASLA:TAX:LDY#15
 540 .nextletter LDAnames,X:CMP#13:B
EQskip:JSR&FFEE:INX:DEY:BPLnextletter
 550 .skip PLA:TAX
 560 INX:CPX#9:BCCnextscore
 570 RTS
 580 .dots LDA#46:.dot1 JSR&FFEE:DEY
:BPLdot1:RTS
 590 .oswordblock OPT FNequb(0)
 600 OPT FNequb(names DIV 256)
 610 OPT FNequb(15)
 620 OPT FNequb(&20)
 630 OPT FNequb(&7E)
 640 .message OPT FNequb(31)
 650 OPT FNequb(3)
 660 OPT FNequb(9)
 670 OPT FNequb(133)
 680 OPT FNequs("You're score is in
the top nine")
 690 OPT FNequb(31)
 700 OPT FNequb(7)
 710 OPT FNequb(11)
 720 OPT FNequb(134)
 730 OPT FNequs("Please enter your n
ame")
 740 OPT FNequb(31)
 750 OPT FNequb(11)
 760 OPT FNequb(13)
 770 OPT FNequb(130)
 780 OPT FNequb(0)
 790 .message2 OPT FNequb(31)
 800 OPT FNequb(6)
 810 OPT FNequb(23)
 820 OPT FNequb(133)
 830 OPT FNequs("Press a key to cont
inue")
 840 OPT FNequb(0)
 850 ]NEXT
 860 $newscore="    4560"
 870 $scores="    9000    8000    70
00    6000    5000    4000    3000
  2000    1000"
 880 $names="HISSING SID     "+"FRED
DY FLUKE     "+"KEYBOARD BASHER "+"SQU
IDGY BOD      "+"KONG BASHER     "+"TR
ICKY DICKY     "+"SPARROW MAN     "+"H
AVE A NICE DAY "+"DENIM DENNY     "
 890 CALL start
 900 END
 910 DEFFNequb(byte)
 920 ?P%=byte
 930 P%=P%+1
 940 =pass
 950 DEFFNequs(string$)
 960 $P%=string$
 970 P%=P%+LEN($P%)
 980 =pass
```

Program II

From Page 145

block at location &A60 with the following information:

Input buffer address = &920
Line length = &20 (32 characters)
Minimum Ascii = &20 (32 decimal)
Maximum Ascii = &7E (126 decimal)

Therefore the corresponding control block would be:

&A60 = &20
&A61 = &9
&A62 = &20
&A63 = &20
&A64 = &7E

At this point the OSWORD routine can be called.

The micro will now accept an input. Pressing Return or Escape will terminate the routine. Ctrl+U will delete the whole line, Delete will erase the previous character and Copy will copy the character at the editing cursor. On exit the carry flag will be 1 if Escape terminated the input.

Any attempt to enter more characters than specified in the control block will result in a beep – VDU 7.

Characters which are outside the Ascii range are printed but not entered into the string buffer.

For those of you who are interested, Basic uses OSWORD 0 for its INPUT statements.

Another input routine, used in line 390, is OSRDCH (&FFE0). This waits until a key is pressed and returns the key's Ascii value in the accumulator – just like Basic's GET command.

If Escape is pressed the accumulator will contain &1B. This must be acknowledged by issuing OSBYTE &7E (126) – also on line 390.

● *That's it for this month. Next month we'll look at random numbers and other useful techniques used in machine code games.*

50-70	Define work space area for the high scores.
80-100	Enter assembler.
110	Changes to Mode 7 – VDU 22,7.
120-180	Check if the new score is greater or equal to the other high scores.
190-200	Print out a message indicating that the new score is in the top 9.
210	Retrieves the position in the table where the new score is to be inserted.
220	Multiplies the position by 8 to get the offset for the corresponding score.
230	Multiplies again by 2 to get the offset for the corresponding name.
240	If the new score is at the bottom of the high score table skip the insert routine – it's not necessary.
250-300	Move all of the names and scores down one place in the table.
310-320	Copy the new score into the score table.
330-340	Clear the existing name from the table.
350	Accepts a name for the new score.
360	Calls the subroutine which displays the high score table.
370-380	Output the message "Press a key to continue".
390	Calls the read character routine and clears the Escape condition.
400-570	High score display routine.
400	Clears the screen – VDU 12.
410	Loads the X register with 0 – this is used to keep count of the score being displayed.
420	Positions the cursor – VDU31,1,Y*3+2, where Y is the position in the table of the score.
430-440	Output a colour control character.
450-460	Output a flashing control character if the score being displayed is the new score.
470	Prints the position of the score 1,2..9.
480	Prints four dots.
490	Finds the offset of the score's start position and places it in X.
500	Prints the score.
510	Prints five dots.
520-530	Find the offset of the name's start position and place it in X.
540	Prints the names out.
550-560	Test if all nine scores have been displayed.
570	Exit routine.
580	Prints out dots. The number is determined by the Y register.
590-630	OSWORD 0 control block.
640-780	Text for new high score message.
790-840	Text for "Press a key to continue".
850	Exits assembler.
860	Places " 4560" into the "newscore" work space – the spaces are very important.
870	Sets up the high scores.
880	Sets up the high score names.
890	Calls the routine.
900	Exits program.
910-940	Macro for Equate byte (EQUB).
950-980	Macro for Equate string (EQUS).

Description of Program II

MACOB

MAIL ORDER WITH A DIFFERENCE

SYSTEMS

DIRECT TO YOU AT WHOLESALE PRICES

COMPUTERS

BBC B Plus	£489
BBC-B	£329
BBC-BD	£419
DFS (Acorn)	£90
Z80 2nd Processor	£389
6502 2nd Processor	£189
CBM 64	Available for
Sinclair	export

PRINTERS

Canon PW1080A	£279
Canon PW1156A	£389
Kaga KP810	£279
Epson RX80F/T	£232
Quendata 20cps	£239
Juki 6100	£332

FIRMWARE

View	£48
Wordwise +	£48
Disc Doctor	£29
Printmaster	£29

DATA RECORDERS

Omega	£21

MONITORS

Microvitec 1431	£185
Microvitec 1451	£279
Philips BM7502	£83

DISKS

D/S D/D 80T	£20 (box of 10)

(Fully guaranteed)
box of 10

MODEMS

Pace Nightingale	£130

DISK DRIVES

Cumana CSX100 SS 40T	£103
Cumana CSX400 DS 40/80T	£155
Cumana CD200 SS 40T + PSU	£242
Cumana CD800 DS 40/80T + PSU	£356
PSU	£20

- Price includes VAT at 15% and delivery is FREE in the UK.
- To order just send a cheque or postal order to us and the goods reach you fast.
- Trade enquiries welcome.
- Remember, our name is our guarantee.

MACOB SYSTEMS

8 Lister Gardens, London N18 1HZ. Tel: 01-803 1622
Telex: 8954029 att.MACOB

Are YOU a first rate PROGRAMMER?

Then join the Professionals!

Award-winning Database Software needs more programmers, both for freelance work and permanent positions.

Applicants must be fluent in both Basic and machine code on at least one of the popular micros, and preferably have experience of others.

Experience in the software industry is not essential, but obviously candidates must have written good quality software in the past and samples will be required.

Pay is negotiable, depending on age, experience and qualifications. There are excellent prospects for hard working, skilful programmers.

Please send SAE, CV and an example of your work which will be returned uncopied to:

Peter Davidson, Software Manager, Database Software, Europa House, 68 Chester Road, Hazel Grove, Stockport SK7 5NY.

Concluding

HOW TO WRITE MACHINE CODE GAMES

By KEVIN EDWARDS

RANDOM THOUGHTS and KEYBOARD CAPERS

IN this, the final part of the series, we shall be looking at random numbers and routines to check the keyboard. We'll start off with random numbers.

There are very few games which don't use random numbers. So for this reason we shall see how random numbers can be created using several different methods.

In BBC Basic, random numbers are created by applying a complex algorithm on a series of numbers, known as the random seed. This causes pseudo random numbers to be set up in the seed. These seem to have no set pattern and so are considered random. In fact, they follow a series which is very complex.

The complexity is such that the series repeats only after several million calls!

BBC Basic's random number generator is seeded with the same number each time the micro is turned on. This results in the same random numbers being generated. If you don't believe me, turn your micro off, then on, and run the following program:

```
10 FOR L=1 TO 10
20 PRINT RND(10)
30 NEXT
```

Repeating the process will result in the same numbers being generated.

The only way to stop the numbers repeating is to initialise the seed with a different value.

The seed used by Basic can be found in zero page locations &D to &11. One obvious way to change the seed is to poke new values into these locations.

The problem with poking values into the seed is that these new numbers must be random else another series of numbers will be repeated.

We now need random numbers to seed the random number generator. We seem to be going backwards don't we? In fact, we can seed the random number generator by issuing an RND command with a negative parameter.

This is favoured to the naughty poking method. So a command such as:

```
dummy=RND(-TIME)
```

at the start of the program would provide a solution.

Notice that TIME was used to reset the seed. This is quite acceptable because its value will almost certainly be different each time it is executed since TIME is always changing. This results in an almost perfectly randomised seed.

Implementing this in machine code is quite a tricky business. Before we do this, let's see how BBC Basic creates a new set of values in the seed by disassembling the Basic ROM.

The start address of the routine is &AF87 for Basic II micros or &AFB6 for Basic I. If you don't have access to a disassembler, printed above is the ROM routine which is, of course, the copyright of Acorn.

```
.start  LDY #&20
.next   LDA &F
        LSR A
        LSR A
        LSR A
        EOR &11
        ROR A
        ROL &D
        ROL &E
        ROL &F
        ROL &10
        ROL &11
        DEY
        BNE next
        RTS
```

Calling *start* creates five random bytes in locations &D to &11 – the seed workspace. (Assuming the locations have been seeded with a value not equal to zero.)

The routine looks simple but is in fact quite complex. See if you can work out what happens to the seed bytes.

Program I is a fast, crude random number routine which repeatedly creates a single random number and prints it out. Press Ctrl N before running the program to turn on the paging mode.

The routine creates a random byte by accessing several very useful locations. These are:

&FE44 – Timer 1 low order counter in

```
10 REM Crude random number
20 REM generator
30 REM By Kevin Edwards
40 FOR pass=0 TO 2 STEP 2
50 P%=&C00
60 [OPT pass
70 .start
80 LDA&FE44:EOR&FC:EOR&FE65
90 JSRprintbyte
100 LDA#ASC" "
110 JSR&FFEE:JSR&FFEE
120 JMPstart
130 .printbyte
140 PHA
150 LSRA:LSRA:LSRA:LSRA
160 JSRdigitout
170 PLA:AND#&F
180 .digitout
190 CMP#10:BCCnumeric
200 ADC#6
210 .numeric
220 ADC#48
230 JMP&FFEE
240 ]NEXT
250 CALLstart
```

Program I

From Page 81

the system VIA (Versatile Interface Adapter). This counter is decremented every millionth of a second and so provides a reasonably random number.

&FC – Accumulator copy from the last interrupt. Each time an interrupt occurs the accummulator is saved in location &FC. Since interrupts can occur at almost any time the accumulator's value will be almost unpredictable – depending upon the section of program being executed when the interrupt is requested.

&FE65 – Timer 1 high order counter in the user VIA. Again, this decrements quickly but obviously at a slower rate than the low order counter – 1/256th of the speed.

Reading from &FE44 and EORing with the other two locations results in a random number. This method is very crude and probably has a poor frequency distribution, but it isn't half fast!

A word of warning. The VIA registers cannot be directly accessed if the program is being executed in the 6502 2nd processor memory.

Also included in Program I is a hex number print routine. This prints the contents of the accumulator as a 2 digit hex number. You'll find this routine very useful if you haven't already written a similar one.

For cases where a "better" random number generator is needed, Program II should be used. Although it's relatively slow, it provides five random bytes.

The algorithm Program II uses is similar to the one in the Basic ROM but uses a slightly different approach.

Line 360 in Program II is used to initialise the seed bytes. As you can see, it uses a low order counter in the user VIA as a seeding value.

The best way to generate a true random set of numbers is to use the random routine from Program I to initialise the seed for Program II, then Program II's random number generator can be used to create the random bytes.

The end result will be a reliable set of random values.

For the majority of cases random bytes in the range 0-255 are quite acceptable. But there are times when custom random number routines are needed.

For example, in a dice game we require random numbers between 1 and 6. Obviously a random number between 0 and 255 is of no use in this case.

Unfortunately there is no simple answer to this problem. The only easy way to do this is by using floating point numbers.

Let me explain.

A random integer between 1 and n can be created by using the following statement:

```
rand=INT(RND(1)*n)+1
```

Which is usually shortened to:

```
rand=RND(n)
```

The RND(1) returns a number between 0 and 0.999999999 and the *n* enlarges this number so that it falls within the range we require. The integer part of the result is taken and 1 is added to it to give us our final answer.

Doing all this in machine code is quite a task. First of all a random number between 0 and 0.999999999 must be created (not too difficult). This is then multiplied by *n* (don't forget, we're in machine code). After which the integer part of the result is taken and 1 is added to give us our final answer.

Let me warn you now, machine code floating point routines are not easy to write. We could of course use the Basic ROM routines, but that's cheating!

You'll find that the Basic ROM contains some clever, and very useful routines which are needed at some stage by all machine code programmers.

It is at this point that praise should be given to the author of BBC Basic, Roger Wilson (and all others concerned). By using very clever programming skills he provided us with one of the fastest, comprehensive versions of Basic available for any home micro. And to add the icing to the cake, he included a superb assembler.

That's enough of random numbers. Next we'll see how to check the keyboard.

Checking if a key is pressed is quite a simple process – all we do is call a routine in the operating system – OSBYTE (&FFF4).

Before this can be done the A, X and Y registers must be set up with various information.

The accumulator should contain &81 indicating that OSBYTE call &81 (read

```
10 REM Random number generator
20 REM By Kevin Edwards
30 seed=&70
40 FOR pass=0 TO 2 STEP 2
50 P%=&C00
60 [OPT pass
70 .random LDY#4
80 .again
90 LDAseed,Y:ADCseed+2:EORseed+1
100 LDX#4
110 .again2
120 ROLseed+2:EORseed,X:SBCseed,Y
130 SBCseed,X:ASLA:RORseed,X
140 DEX:BPLagain2
150 DEY:BNEagain
160 LDX#4
170 .nextseed
180 LDAseed,X:JSRprintbyte
190 LDA#ASC" "
200 JSR&FFEE:JSR&FFEE:JSR&FFEE
210 DEX:BPLnextseed
220 RTS
230 .printbyte
240 PHA
250 LSRA:LSRA:LSRA:LSRA
260 JSRdigitout
270 PLA:AND#&F
280 .digitout
290 CMP#10:BCCnumeric
300 ADC#6
310 .numeric
320 ADC#48
330 JMP&FFEE
340 ]NEXT
350 FORL%=0TO4
360 L%?seed=?&FE44
370 NEXT
380 REPEAT
390 CALLrandom
400 UNTIL 1=2
```

Program II

a key) is being executed.

The X register indicates the key being checked. Its value should correspond to the key's negative key number. These can be found on page 275 of the User Guide.

Once the negative value is known it must be converted into a 2's complement byte. This is done by printing the hexadecimal value of the number. For example, to find the value for X when we wish to check for the Spacebar we would enter:

```
PRINT ~-99
```

which would give:

```
FFFFFF9D
```

The only part of the number we're interested in is the 9D – not the six leading Fs. So to check for Space we load the X register with &9D.

The Y register must always contain &FF.

Our key read routine would look something like this:

```
.check_key      LDA #&81
                LDX #&9D
                LDY #&FF
                JSR &FFF4
```

On exit the X and Y register will contain &FF if the key is pressed. If Y is 0, the key isn't pressed. Adding this information to the routine would result in:

```
.check_key      LDA #&81
                LDX #&9D
                LDY #&FF
                JSR &FFF4
                CPY #&FF
                BNE not_pressed
                JSR reaction
.not_pressed    (Rest of program)
```

To save repeating code a general keyboard routine can be used. For this routine only the X register is required:

```
.check_key      LDA #&81
                LDY #&FF
                JSR &FFF4
                CPY #&FF
                RTS
```

On exit the Z flag is 1 if the key is pressed or 0 if it isn't. This can be checked for using BEQ or BNE in the following way:

```
.check_space    LDX #&9D
                JSR check_key
                BNE check_return
                JSR reaction1
.check_return   LDX #&B6
                JSR check_key
                BNE check_space
                JSR reaction2
                JMP check_space

.reaction1      (routine if space
                is pressed)
                RTS

.reaction2      (routine if return
                is pressed)
                RTS
```

And that concludes the series. I hope you've enjoyed following it as much as I've enjoyed writing it.

Over the past few months we've covered only a fraction of the things associated with machine code games.

I hope you'll remember me when you're earning thousands of pounds for your latest machine code game. See you again some time.

MAXIM

A NEW 16K EPROM

- Write in machine code or in program form.
- Assembles either from disc or directly from keyboard into memory. Only disc assembly requires two passes. Both methods can produce machine code or program format.
- Instructions and labels in program form may be deleted, added to, inserted or altered. JMP tables are specially maintained.
- Conversion from program to machine code takes typically 2 seconds for 16k of code. Subsequently only limited alterations may be made.
- Optional RUN time filing system automatically saves the two most recent copies of the program.
- Display always shows eighteen lines with the operational line just over half way down the screen.
- Display moves forwards or backwards. Maxim calculates the correct starting address.
- Labels may be used and saved with or without a program.
- Program and labels are displayed in ASCII assembler form with addresses. If needed, programmes may be *SPOOLed into ASCII files suitable for immediate disc assembly.
- Comprehensive set of debugging utilities. Maxim may be made fully transparent for single stepping both ROM and RAM. There is a second screen (all graphics modes). BASIC II can be freely used from within Maxim.
- Maxim works at C000 on the 6502 Second Processor, leaving the 'language' area free from 8000 BFFF. It is transparent for writing programs intended to be eprommed.

Maxim Computer Systems Ltd.
17 Wigmore Street, London W1

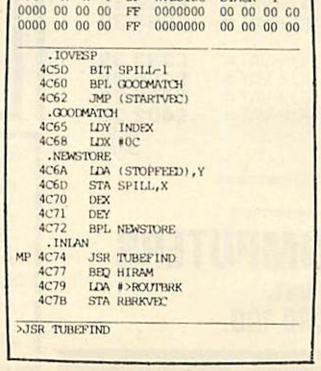
The friendliest assembly system in the business – makes assembler almost as easy as BASIC!

Tired of the hours wasted peering cross-eyed at endless addresses, trying to remember what they mean?
Maxim displays the program just as you wrote it. Makes debugging a pleasure – while it lasts!

AND you can write programs straight into memory – complete with labels!

£65 + VAT (includes p&p)

Programming Screen

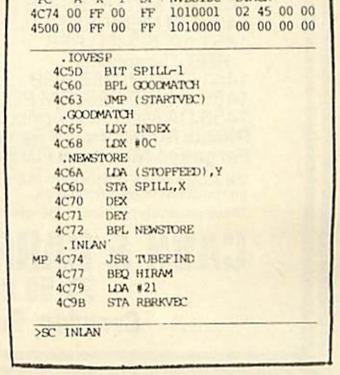


```
 PC   A  X  Y  SP  NVBDIZC  STACK  I
0000 00 00 00  FF  0000000  00 00 00 C0
0000 00 00 00  FF  0000000  00 00 00 00

   .IOVESP
  4C5D   BIT SPILL-1
  4C60   BPL GOODMATCH
  4C62   JMP (STARTVEC)
   .GOODMATCH
  4C65   LDY INDEX
  4C68   LDX #0C
   .NEWSTORE
  4C6A   LDA (STOPFEED),Y
  4C6D   STA SPILL,X
  4C70   DEX
  4C71   DEY
  4C72   BPL NEWSTORE
   .INLAN
MP 4C74   JSR TUBEFIND
  4C77   BEQ HIRAM
  4C79   LDA #>ROUTBRK
  4C7B   STA RBRKVEC

>JSR TUBEFIND
```

Debugging Screen



```
 PC   A  X  Y  SP  NVBDIZC  STACK
4C74 00 FF 00  FF  1010001  02 45 00 00
4500 00 FF 00  FF  1010000  00 00 00 00

   .IOVESP
  4C5D   BIT SPILL-1
  4C60   BPL GOODMATCH
  4C63   JMP (STARTVEC)
   .GOODMATCH
  4C65   LDY INDEX
  4C68   LDX #0C
   .NEWSTORE
  4C6A   LDA (STOPFEED),Y
  4C6D   STA SPILL,X
  4C70   DEX
  4C71   DEY
  4C72   BPL NEWSTORE
   .INLAN'
MP 4C74   JSR TUBEFIND
  4C77   BEQ HIRAM
  4C79   LDA #21
  4C9B   STA RBRKVEC

>SC INLAN
```

MACHINE CODE GAMES

By KEVIN EDWARDS

Now let's put it all together

THERE's been such a tremendous response to my machine code games series that the editor's decided that this month I've to give you a complete game combining several routines from the series. This, I hope, will help you understand how a machine code game should be written.

The game uses two sprites, a character we'll refer to as a ball, and an arrow. Your aim is to fire the arrow at the right moment so that it collides with the moving ball – it couldn't be simpler.

The ball drops down the screen from a random column position at different speeds. The arrow is controlled by you and can only be fired across the screen from left to right – by pressing the Spacebar. It's not exactly Elite, but at least I can explain how it works. The program uses two main subroutines which are both related to sprites.

The first routine *sprite* is responsible for displaying and erasing sprites – see Page 92 of the June, 1985, issue of *The Micro User* for a thorough description.

The other routine *calc_loc* calculates the screen location for a screen X and Y coordinate – the origin of which is the bottom left corner. This was first used on Page 93 of the May, 1985, issue of *The Micro User*.

There are several other subroutines which are responsible for simple tasks such as setting up information for the sprite routines and generating random numbers.

The object code created by the program is assembled at location &2E00 and occupies around 400 bytes of memory. The game can be executed, once assembled, by entering:

```
CALL start
```

An arrow will appear at the bottom left of the screen. You must now wait for the ball – a large circle with legs and an ugly face – to appear in the top half of the screen. And at the right moment you should fire the arrow, by pressing Space, so as to cause a collision.

Here's an outline of the steps we follow to achieve this:

1. Initialise various variables and reset the sprites.
2. Wait for a random length of time and then put the ball on the screen.
3. Move the ball a small distance down the screen.
4. If the arrow is in motion move it right by one place. Otherwise, check if the spacebar is pressed. If it is, move the arrow.
5. Check if the two sprites have collided. If they have, make a beep and jump to step 1.
6. Check if either sprite has reached the edge of the screen. If this is the case jump to step 1, otherwise, jump to step 3.

As you can see, it's quite simple. Let's take a look at how the program achieves this.

All this may seem a long-winded way to perform such a simple task, but you'll

→

10-120	Initialise various variables and select the assembler options.
130-310	The sprite routine. This is responsible for displaying/erasing sprite characters on/from the screen.
320-400	Calculate the screen location for a given X,Y screen coordinate. The origin is taken to be at the bottom left of the screen.
420	Resets the arrow's status. If the location *moveflag* contains zero the arrow is stationary, otherwise, the arrow is moving.
430-440	Randomly select either 2 or 3 and store it in location *speed*. This indicates the number of pixels by which the ball will move down the screen. A large value would make the ball move faster than a smaller one.
450-460	Initialise the arrow's screen coordinates to the bottom left of the screen.
470	Displays the arrow on the screen.
480	Places a random nybble, a four bit number between 0 and 15, into the X register.
490-530	Wait for a random length of time. The length is determined by the random number returned from line 480. The delay is achieved by having three nested loops. This can take a long time to execute even in machine code. The random number previously created is used for the outer loop.
540-550	Calculate a random X coordinate between 30 and 60 – in steps of 2 – for the ball. Giving the ball a random start address makes it more difficult to play the game.
560	Resets the ball's Y coordinate to 200 – in the top half of the screen.
570	Displays the ball on the screen.
590	Checks if the arrow is moving. If it is then branch. This is done to avoid the key check routine which isn't needed because the arrow is already moving.
600	Tests to see if the Spacebar is pressed. Branch if false.
610	Stores &FF in the arrow movement flag. This indicates that the arrow is now in motion.
620	Erases the arrow from the screen.
630	Increments the arrow's X coordinate to move it right one place.
640	Checks if the arrow has reached the right-hand edge of the screen and branches if false.
650	Removes the ball from the screen then jumps back to the start of the main loop.
660-670	Test for a collision between the two sprites. This is done by peeking the screen memory

→

```
   10 REM Simple demo game
   20 REM By Kevin Edwards
   30 MODE 2
   40 swidth=&74
   50 sheight=&75:sheight2=&78
   60 coltoplow=&76:screenlow=&70
   70 coltophigh=&77:screenhigh=&71
   80 temp0=&80:temp1=&81
   90 temp2=&82
  100 HIMEM=&2E00
  110 FORpass=0TO2STEP2:P%=HIMEM
  120 [OPTpass
  130 .sprite STXswidth
  140 STYsheight:STYsheight2
  150 .user_entry LDX#0
  160 .main_part LDAcoltoplow:AND#&F8
:STAscreenlow
  170 LDAcoltophigh:STAscreenhigh
  180 LDAcoltoplow:AND#7:TAY
  190 .column LDA&FFFF,X:EOR(screenlo
w),Y
  200 .onto_screen STA(screenlow),Y
  210 INX:BEQ inc_data_high
  220 .end_checks INY:CPXsheight2:BEQ
end_of_column
  230 CPY#8:BNEcolumn
  240 LDAscreenlow:ADC#&7F:STAscreenl
ow
  250 LDAscreenhigh:ADC#2:STAscreenhi
gh
  260 LDY#0:BEQ column
  270 .inc_data_high INCcolumn+2:JMPe
nd_checks
  280 .end_of_column CLC:LDAcoltoplow
:ADC#8:STAcoltoplow:BCCno_high:INCcol
tophigh
  290 .no_high CLC:LDAsheight2:ADCshe
ight:STAsheight2
  300 DECswidth:BNEmain_part
  310 RTS
  320 .calc_loc LDA#&30:STAtemp1
  330 LDA#0:STAtemp2
  340 TYA:EOR#&FF:TAY:AND#7:STAtemp0
  350 TYA:LSRA:LSRA:LSRA:ASLA:TAY
  360 TXA:ASLA:ROLtemp2:ASLA:ROLtemp2
  370 ASLA:ROLtemp2
  380 ADCtemp0:ADC&C376,Y:STAcoltoplo
w
  390 LDAtemp2:ADCtemp1:ADC&C375,Y:ST
Acoltophigh
  400 RTS
  410 .start
  420 LDA#0:STAmoveflag
  430 JSRrandombyte:AND#1
  440 CLC:ADC#2:STAspeed
  450 LDA#0:STAarrowX
  460 LDA#30:STAarrowY
  470 JSRarrowonscreen
  480 JSRrandnyb
  490 LDY#0
  500 .delay1
  510 DECtemp0:BNEdelay1
  520 DEY:BNEdelay1
  530 DEX:BPLdelay1
  540 JSRrandombyte:AND#&F:ASLA
  550 CLC:ADC#30:STAballX
  560 LDA#200:STAballY
  570 JSRballonscreen
  580 .repeat1
  590 BITmoveflag:BMImoveacross
  600 LDX#&9D:JSRinkey:BNEnotspace
  610 LDA#&FF:STAmoveflag
  620 .moveacross JSRarrowonscreen
  630 INCarrowX
  640 LDAarrowX:CMP#70:BNEreplacearro
w
  650 .eraseball JSRballonscreen:JMPs
tart
  660 .replacearrow LDXarrowX:LDYarro
wY:JSRcalc_loc
  670 LDY#&40:LDA(coltoplow),Y:BEQnoh
it
  680 LDA#7:JSR&FFEE
  690 JSRballonscreen:JMPstart
  700 .nohit JSRarrowonscreen
  710 .notspace LDA#19:JSR&FFF4
  720 JSRballonscreen
  730 LDAballY:SEC:SBCspeed:STAballY
  740 CMP#10:BCSreplaceball
  750 JSRarrowonscreen:JMPstart
```

From Page 83

find this is the case for most machine code games.

When writing your own games you should break the program into simple steps – as I've done at the start of the article. This will make the programming easier and allow you to identify sections of repeated code which can be put into a subroutine – thus saving you memory.

I hope this has given you some more ideas for writing your own games. For starters, try modifying the one described here. For instance, you might have two arrows, one low, the other high. Or more than one ball could be dropping at the same time. Or the ball could zig-zag on its way down. And you might like to give it some sound effects. Try it – you'll learn a lot by tinkering!

And if you come up with anything interesting, send it in and let us have a look.

	&40 bytes ahead of the arrow's top left screen location. If the screen byte returned is zero, indicating no collision, the branch is taken, otherwise, a collision has occured.
680	Makes a beep – VDU 7 – indicating a collision.
690	Erases the ball from the screen and starts the program again.
700	Puts the arrow on the screen in its new position.
710	Waits for next screen re-fresh – OSBYTE 19. This gives smooth movement and reduces flicker.
720	Deletes the ball from the screen.
730	Decreases the ball's Y coordinate to move it down the screen. The number subtracted depends on the contents of the location given by the variable *speed*. This will be either 2 or 3 depending on the number stored there at the start of the main loop – see line 440.
740	Checks if the new Y coordinate is greater or equal to 10. If this is true the branch is taken because the ball has not gone off the bottom of the screen.
750	Deletes the arrow from the screen then jumps back to the start of the program.
760-770	Display the ball in its new position then jump to start of main loop.
780-820	Display/erase the arrow on/from the screen. Before the sprite routine can be called various information must be set up. This includes the graphic data pointer – 790-800 – the screen address – 810 – and the sprite's dimensions – 820.
830-870	Display/erase the ball on/from the screen. See the previous description for additional information.
880-890	Check if a certain key is pressed. On entry the X register should contain the negative inkey number of the key to be tested. On exit the Z flag will be 1 if the key is pressed and 0 if it's not.
900	Generates a random nybble in the X register.
910	Generates a random byte in the Accumulator.
920-970	Reserve room for various variables used by the routine.
990-1010	Read the sprite graphic data into page &C.
1 030-1110	The arrow graphic data.
1 120-1360	The ball graphic data.

```
 760 .replaceball JSRballonscreen
 770 JMPrepeat1
 780 .arrowonscreen
 790 LDA#0:STAcolumn+1
 800 LDA#&C:STAcolumn+2
 810 LDXarrowX:LDYarrowY:JSRcalc_loc
 820 LDX#8:LDY#8:JMPsprite
 830 .ballonscreen
 840 LDA#&40:STAcolumn+1
 850 LDA#&C:STAcolumn+2
 860 LDXballX:LDYballY:JSRcalc_loc
 870 LDX#8:LDY#24:JMPsprite
 880 .inkey LDA#&81:LDY#&FF
 890 JSR&FFF4:CPY#&FF:RTS
 900 .randnyb JSRrandombyte:AND#&F:T
AX:RTS
 910 .randombyte SEI:LDA&FE68:EOR&FE
44:EOR&FC:CLI:RTS
 920 .arrowX NOP
 930 .arrowY NOP
 940 .ballX NOP
 950 .ballY NOP
 960 .moveflag NOP
 970 .speed NOP
 980 ]NEXT
 990 FOR L%=0 TO &FF
1000 READ L%?&C00
1010 NEXT
1020 END
1030 REM Arrow data
1040 DATA&F,0,3,&3F,&3F,3,0,&F
1050 DATA5,&A,3,&3F,&3F,3,&A,5
1060 DATA&A,&A,3,&3F,&3F,3,&A,&A
1070 DATA0,0,3,&3F,&3F,3,0,0
1080 DATA0,0,3,&3F,&3F,3,0,0
1090 DATA1,0,3,&3F,&3F,3,0,1
1100 DATA&2B,&17,3,&3F,&3F,3,&17,&2B
1110 DATA0,2,&2B,&3F,&3F,&2B,2,0
1120 REM Ball data
1130 DATA0,&14,&14,&14,&14,&14,&3C,&
3C
1140 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1150 DATA&3C,&3C,&14,&14,&14,&11,&33
,&33
1160 DATA&14,&3C,&3C,&3C,&29,&29,&29
,&3C
1170 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1180 DATA&3C,&3C,&33,&33,&36,&36,&33
,&33
1190 DATA&3C,&3C,&3C,&3C,3,3,3,&16
1200 DATA&3C,&3C,&34,&34,&30,&30,&30
,&38
1210 DATA&3C,&3C,&36,&36,&3C,&3C,&3C
,&14
1220 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1230 DATA&3C,&3C,&3C,&3C,&3C,&3C,&30
,&30
1240 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1250 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1260 DATA&3C,&3C,&3C,&3C,&3C,&3C,&30
,&30
1270 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1280 DATA&3C,&3C,&3C,&3C,3,3,3,&29
1290 DATA&3C,&3C,&38,&38,&30,&30,&30
,&34
1300 DATA&3C,&3C,&39,&39,&3C,&3C,&3C
,&28
1310 DATA&28,&3C,&3C,&3C,&16,&16,&16
,&3C
1320 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1330 DATA&3C,&3C,&33,&33,&39,&39,&33
,&33
1340 DATA0,&28,&28,&28,&28,&28,&3C,&
3C
1350 DATA&3C,&3C,&3C,&3C,&3C,&3C,&3C
,&3C
1360 DATA&3C,&3C,&28,&28,&28,&22,&33
,&33
```

This listing is included in this month's cassette tape offer. See order form on Page 173.

DISC DRIVES

D80 SMP	Dual Drive (2×400K) 40/80 switchable, double sided + Monitor Stand + PSU	**£320 inc VAT (b)**
S80SP	Single Drive (400K) 40/80 switchable double sided + PSU	**£160 inc VAT (b)**
S80S	Single Drive (400K) 40/80 switchable double sided	**£120 inc VAT (b)**
S80FD	Single Drive (400K) Full height 80 track double sided	**£100 inc VAT (b)**
S80F	Single Drive (200K) 80 track single sided	**£80 inc VAT (b)**
S40S	Single Drive (100K) 40 track single sided	**£100 inc VAT (b)**
D35	Dual 3.5" drives (800K) 80 track double sided	**£200 inc VAT (b)**
S35	Single 3.5" drive (400K) 80 track double sided	**£115 inc VAT (b)**

All drives fully cased and cabled.

Acorn Interface £55 inc VAT (c)
Acorn DNFS £75 inc VAT (c)
8271 £46 inc VAT (a) 27128 £5 inc VAT (a)
2764 £2.75 inc VAT (a) 6264LP £5 inc VAT (a)

Quantity discounts available

CARRIAGE CHARGES
(a) £1.00 (b) 10.00 (c)£2.00

CARSON DEVELOPMENTS (MU)
84 Highfield Road, Romford, Essex, RM5 3RU
Telephone: 0708 27043

DISKETTES from £10 per pack (ex VAT).

All diskettes are guaranteed and have a lifetime warranty.

	10 of	25 of
40tk ss sd	**11.50**	**27.00**
40tk ss dd	**12.65**	**29.00**
40tk ds dd	**15.00**	**35.00**
80tk ds quad	**17.00**	**41.00**

Prices inc VAT + carriage.

Add £7.00 for a 50 capacity storage box with smoked perspex lid, dividers and non-slip rubber feet.

Datalife Verex

40tk ss sd	–
40tk ss dd	**13.50**
40tk ds dd	**17.50**
80tk ds quad	**23.00**

Sony
3.5" ss dd **£38.00**
3.5" ds dd **£45.00**

Prices inc VAT. Please add £1.00 carriage.

Are you frustrated by page problems, running 2 or more DFS's, ROM interaction, require a single and double density sector editor?
The new SYSTEMS UTILITY ROM from EDSOFT is your answer.
With SYSTEMS UTILITY you can:
Sector edit Acorn, Opus and UDM.
Enable/Disable ROMs – even across control break.
List ROMs.
Check status of ROMs even when disabled.
PEEK and POKE memory.
Display BASIC variables.
Automatically run the BASIC/MC file from disc and leave PAGE at &E00.
Automatically download cassette software from disc.
Display function keys.
Plus numerous other features. **£20 inc VAT (c)**

UDM DDFS 3.0	**£80 inc VAT (a)**
UDM 100% Board	**£17.50 inc VAT (a)**
UDM ROM/16K RAM Expansion Board	**£55 inc VAT (c)**
UDM 3.0 Upgrade	**£5.75 inc VAT (a)**

NEW REPLICA II
Tape to Disc utility for Acorn, Opus + UDM DDFS **£12 inc VAT (a)**

Slimline Drive Case	**£13.80 inc VAT (c)**	Full Height Drives Case	**£18.98 inc VAT (c)**
Single Drive Data Cable	**£6.50 inc VAT (a)**	Single Drive Power Cable	**£3.50 inc VAT (a)**
Dual Drive Data Cable	**£10.50 inc VAT (a)**	Dual Drive Power Cable	**£4.75 inc VAT (a)**